VŠB – Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra informatiky

Mean shift na CUDA

Mean Shift on CUDA

2010 Tomáš Bajar

Prohlašuji, že jsem tuto bakalářskou/diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Rád bych na tomto místě poděkoval svému vedoucímu bakalářské práce Mgr. Štěpánu Šrubařovi za trpělivost, odborné vedení a užitečné rady.

V Ostravě 7. Května 2010 . . . . . .

Abstrakt

Mean Shift je flexibilní segmentační algoritmus, pomocí kterého lze například z obrázku určit jednotlivé části, které patří k sobě. Člověk je schopen tyto části v obrázku rozpoznat sám od sebe, zatímco počítači musíme metodu pro jejich rozpoznání nejprve implementovat, například právě použitím tohoto algoritmu. Časová náročnost výpočtu tohoto algoritmu je ovšem pro nás vysoká díky jeho složitosti, což se negativně promítá do výkonu a tím pádem také do jeho použitelnosti. Tato bakalářská práce si klade za cíl analyzovat stávající algoritmus pro procesor, navrhnout řešení využívající paralelního zpracování pomocí technologie CUDA, implementovat jej a porovnat dosažené výsledky. Práce obsahuje návrh a analýzu takového algoritmu společně s rozborem jeho výsledků.

Klíčová slova: mean shift, CUDA, segmentace

Abstract

Mean Shift is a flexible segmentation algorithm, by which we can for example determine individual parts of a picture that belong together. Man is able to recognize these by himself, while for a computer we must first implement this recognition method, for example by using this algorithm. The time demands for computation of this algorithm are however very big because of its complexity, which negatively affects performance and thus its usefulness. This bachelor work aims to analyze the existing algorithm for processor, suggest a solution using parallel processing by CUDA, implement it and evaluate the accomplished goals. This work consists of design and analysis of such algorithm together with evaluation of its results.

Key words: mean shift, CUDA, segmentation

Seznam použitých zkratek

API – Application Programming Interface

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

GPU – Graphic Processing Unit

Gpix – Giga pixel = 10^9 pixelů

IDE – Integrated Development Environment

MRL – Media Research Lab, laboratoř pro výzkum zpracování multimédií v rámci FEI

PC – Personal Computer

RAM – Random Access Memory

RGB – Red Green Blue

SDK – Software Development Kit

TCP/IP – Transmission Control Protocol / Internet Protocol

TDR – Timeout Detection & Recovery

Obsah

Seznam obrázků

1 Úvod

Člověk se vždy snažil naučit počítač tomu, aby rozuměl zadaným vstupům, ať už se jedná o text, obrázky, video či podobná data. Kdyby počítač rozuměl například tomu, co se nalézá na obrázku, bylo by za tohoto předpokladu relativně jednoduše možné vytvořit nové aplikace pro využití jak v běžném životě, tak v dalších odvětvích lidské činnosti, kde se pro práci využívá přínos počítačů.

Právě toto základní porozumění běžným věcem, které je pro člověka samozřejmostí, je pro počítače v mnoha případech extrémně náročným problémem. Je tedy potřeba počítači od základu naprogramovat pomocí elementárních kroků přesný postup, jak danou problematiku vyřešit. Vzhledem k obrovskému množství možností, které mohou v jednotlivých fázích výpočtu nastat, je nutno neopomenout co nejvíce faktorů, které by ve finále mohly mít vliv na výsledek. Programový kód, který toto všechno zohledňuje, je ovšem většinou velmi komplexní, a tudíž ke svému běhu vyžaduje hodně systémových prostředků.

Při dnešním vývoji moderní techniky nám současné procesory nabízejí stále větší výkon i více jader, které může programátor využít. V poměrně krátkodobém časovém horizontu jsou plánována notná vylepšení procesorů a přidávání dalších jader. Zajímavým fenoménem ovšem začínají být grafické karty, které díky svému rychlému vývoji začínají nabízet alternativu pro obecné výpočty. Při dnešním srovnání grafických karet a procesorů vycházejí grafické karty co do počtu samostatných výpočetních jednotek a celkového výkonu výrazně lépe.

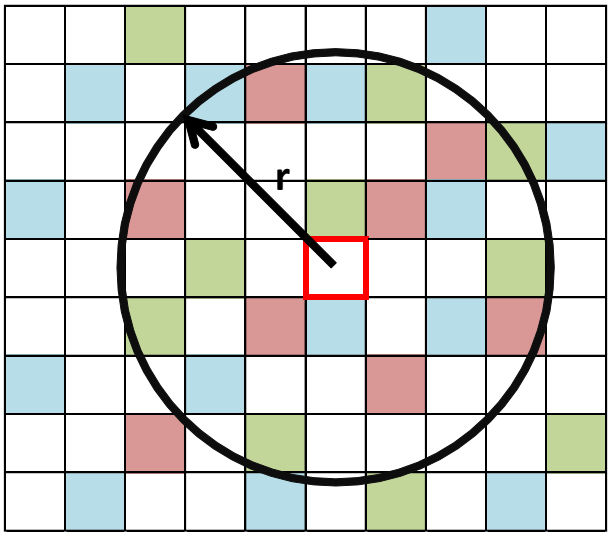
V této práci bude ukázáno, jak problematiku segmentace obrázků zvládá současný dvoujádrový procesor a jak proti němu obstojí moderní grafická karta s podporou technologie CUDA. Práce se rovněž zaměří na rozbor jednotlivých aspektů této problematiky a předvede možnosti nejnovějších technologií firmy NVIDIA.

2 Mean Shift

V této práci se budu zabývat segmentací obrázků. Bude se pracovat s jednotlivými pixely obrázku, které budou reprezentovány svými barevnými složkami. Pro práci s těmito složkami jsem si vybral systém RGB, protože je v praxi velice rozšířený. Obecně by místo těchto tří složek bylo možné použít jakékoli jiné tři barevné kanály. Počet a využití kanálů je v tomto algoritmu také jednoduché změnit. Jednotlivé pixely pro nás budou představovat data, která bude náš algoritmus zpracovávat. Jelikož bylo potřeba použít nějaký obecný algoritmus, který toto dovede, pojednává má práce o metodě Mean Shift.

2.1 Mean Shift obecně

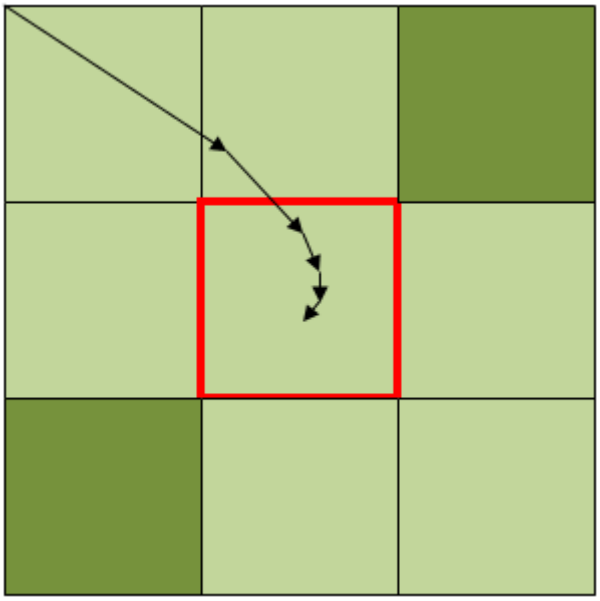
Algoritmus Mean Shift v přesném překladu znamená přesun za středem vzorků. Funguje iterativně, což znamená, že k výsledku se přibližuje postupně v krocích. V každém kroku počítá z výpočetního okna vážený průměr pixelů na základě toho, jak jsou si pixely podobny v jednotlivých barevných složkách, a jak jsou od sebe tyto pixely vzdáleny. Z toho vyplývají základní dva parametry této metody. První z nich je velikost okolí bodu, pro který se algoritmus právě počítá. Druhým parametrem je maximální barevná vzdálenost pixelů výpočetního okna od pixelu středového, pro který výpočet začíná. Pixel výpočetního okna může mít nejvýše tuto hodnotu, aby ještě byl zahrnut do výpočtu jako relevantní.



1 - Příklad výpočetního okna v části obrázku

V každé iteraci hledá algoritmus podle těchto parametrů novou pozici, se kterou bude pracovat v příští iteraci. Nová pozice je vážený střed daného výpočetního okna. Po přesunu na takovou novou pozici je možno spočítat, o jakou vzdálenost se výpočetní okno posunulo. Velikost posunu se směrem k váženému středu všech vzorků bude zmenšovat. Jakmile začnou být posuny velice malé, je velká pravděpodobnost, že se další posuny výpočetního okna nedostanou za hranice aktuálního pixelu a můžeme si proto dovolit výpočet zastavit. Například při počítání s pixely se můžeme dostat na posuny v řádu desetin až setin pixelu, což pro nás přestane být důležité, neboť se iterace již

nedostanou z určeného pixelu. Z uvedeného vyplývá třetí parametr, který je vhodné použít, a to sice minimální délka posunu. Jakmile se tedy iterace posune o méně než tuto hodnotu, ukončíme výpočet pro daný prvek.

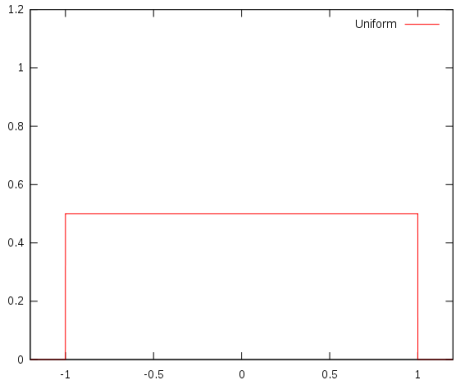


2 - Posun v rámci jednoho vzorku

Po ukončení výpočtu se pozice zastaví v místě největší hustoty vzorků v rámci širšího okolí. Šířka tohoto okolí je dána velikostí poloměru výpočetního okna algoritmu. Čím větší poloměr zvolíme, tím více pixelů z okolí může ovlivnit směr a vzdálenost posunu.

2.1.1 Kernel

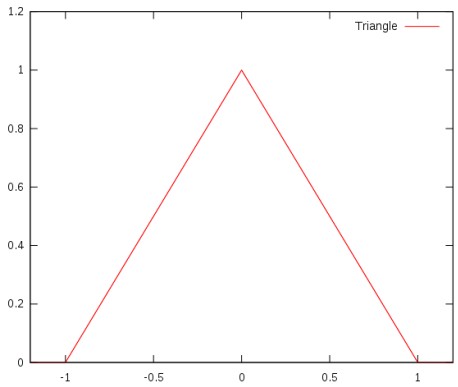
Kernel je radiálně symetrická vážená funkce, která určuje váhy pro jednotlivé pixely výpočetního okna podle jejich vzdálenosti od středu a rozdílu jejich jednotlivých barevných složek. Tato funkce může mít více tvarů, z čehož vyplývá poslední parametr algoritmu Mean Shift. Tento parametr bude určovat tvar kernelu, který se pro výpočet použije. Na následujících obrázcích jsou znázorněny známé tvary, které může kernel mít. U každého je také uvedena rovnice, kterou je definován. [3]



3 - Uniformní kernel

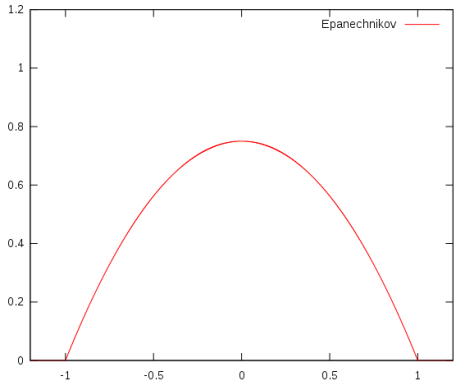
Obrázek 3 představuje uniformní kernel. Tento kernel je nejjednodušší na zpracování, protože v jeho výpočtu není mocnina.

Je definován rovnicí: $K(u) = \frac{1}{2} 1_{\{|u| \leq 1\}}$



4 - Trojúhelníkový kernel

Na obrázku 4 je znázorněn trojúhelníkový kernel definovaný rovnicí: $K(U) = (1 - |u|) 1_{\{|u| \leq 1\}}$

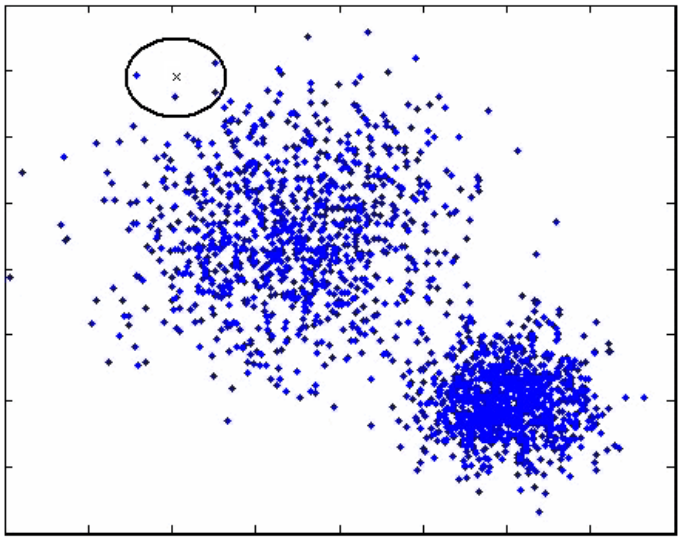


5 - Epanečnikův kernel

Na obrázku 5 je vidět Epanečnikův kernel. Je definován rovnicí: $K(U) = \frac{3}{4}(1 - u^2) 1_{\{|u| \leq 1\}}$

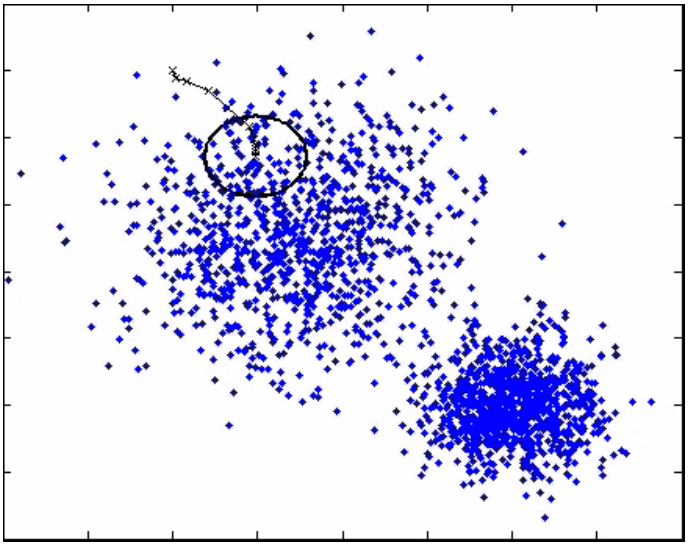
2.2 Ukázka mean shiftu

Hledání největší hustoty vzorků se dá využít k mnoha účelům. Ve své podstatě se tímto algoritmem dají hledat středy množin jakýchkoli prvků. Na obrázku 6 vidíme dvě množiny bodů ve dvourozměrném prostoru, které pro nás představují analyzované vzorky. Ukažme si začátek hledání například na kraji jedné části, kde se vyskytuje hodně vzorků.

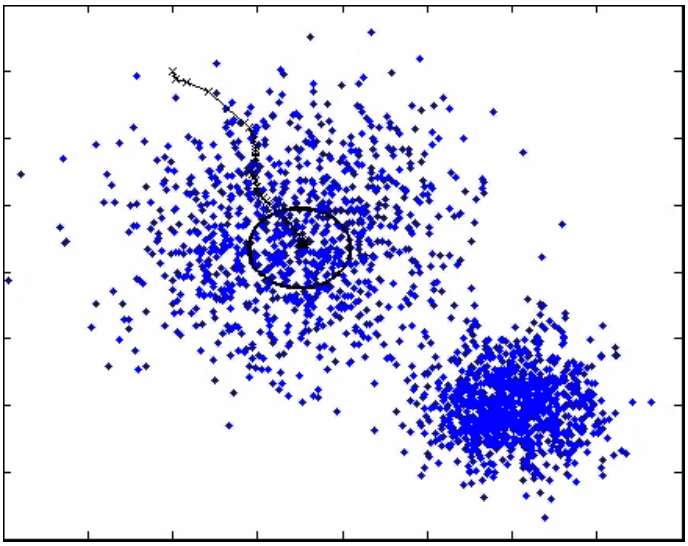


6 - Začátek mean shiftu

Černá kružnice označuje hranice výpočetního okna pro danou iteraci. Algoritmus vypočte v tomto okně novou souřadnici, kterou určí do pravé dolní části okna, a přesune se do ní. Takto se postupně algoritmus přesune až do středu dané části, kde skončí.



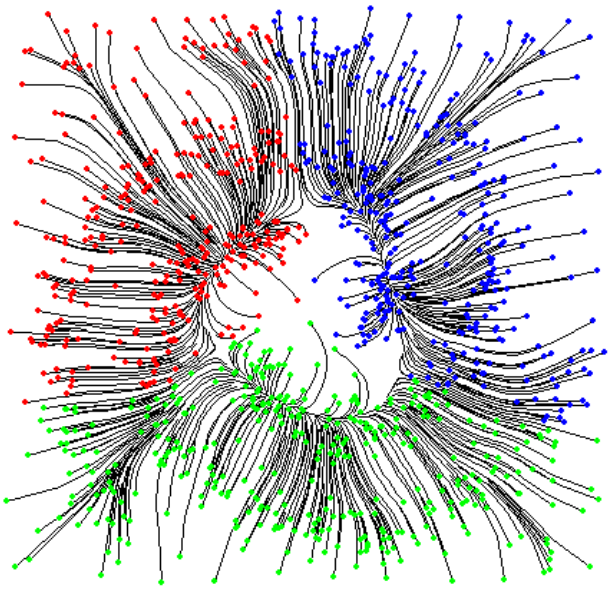
7 - Průběh mean shiftu



8 - Konec mean shiftu

2.3 Segmentace metodou mean shift

Výše uvedeným postupem jsme tedy našli střed vzorků v dané části prostoru. Nyní si představme, že tento postup počítáme pro jeden z prvků této části. Výpočet začne pro okolí vzorku, kdy nalezne všechny vzorky, které náleží do výpočetního okna. Z nich již standardně vypočte střed a postupnými posuny se dostane do stejného místa v prostoru. Takto jsme získali informaci, do které části prostoru daný vzorek patří. Kdybychom toto provedli pro všechny vzorky dané části, výpočet by skončil téměř ve stejném bodě. Následující obrázek 9 ukazuje možné výsledky uvedeného postupu, kde body, označené stejnou barvou, patří do jedné skupiny. Černé čáry znázorňují cestu, kterou se výpočet pro daný bod pohyboval.

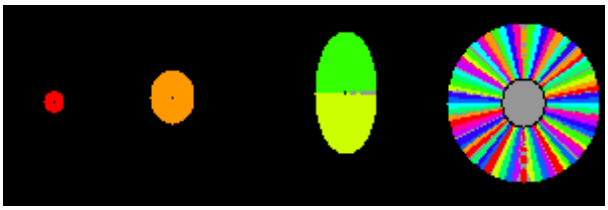


9 - Segmentace vzorků

Při správné reprezentaci těchto poznatků navenek jsme takto schopni určovat jakési celky v prostoru. Můžeme říct, že dané vzorky patří k sobě a tvoří tak jeden segment.

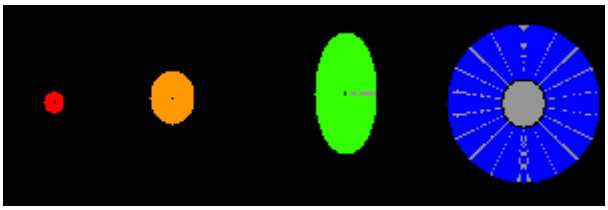
2.4 Mean shift filtrace – sdružování atraktorů

Po dokončení segmentace se výsledek liší podle zadané velikosti výpočetního okna. V mnoha případech se stává, že je část obrázku, kterou se snaží algoritmus určit jako jeden segment, velká natolik, že dojde pouze k částečné segmentaci na více jednotlivých segmentů, které jsou blízko u sebe. Tuto situaci ukazuje následující obrázek 10, kde kruh úplně napravo obsahuje velký počet menších segmentů.



10 - Segmenty s blízkými atraktory

Atraktory, což jsou body, do kterých se pixely jednoho segmentu dostaly, jsou v obrázku označeny černými pixely. V pravém kruhu jsou velice blízko u sebe, což je dáno tím, jak výpočet postupoval z vnějšku směrem dovnitř kruhu. Kdybychom chtěli jednotlivé malé segmenty sloučit do jednoho většího, bylo by třeba sdružit atraktory, které se nacházejí blízko sebe. Obvykle se sdružují atraktory do vzdálenosti velikosti výpočetního okna či části této velikosti. Výsledná segmentace s použitím sdružování atraktorů by v tomto případě vypadala následovně.



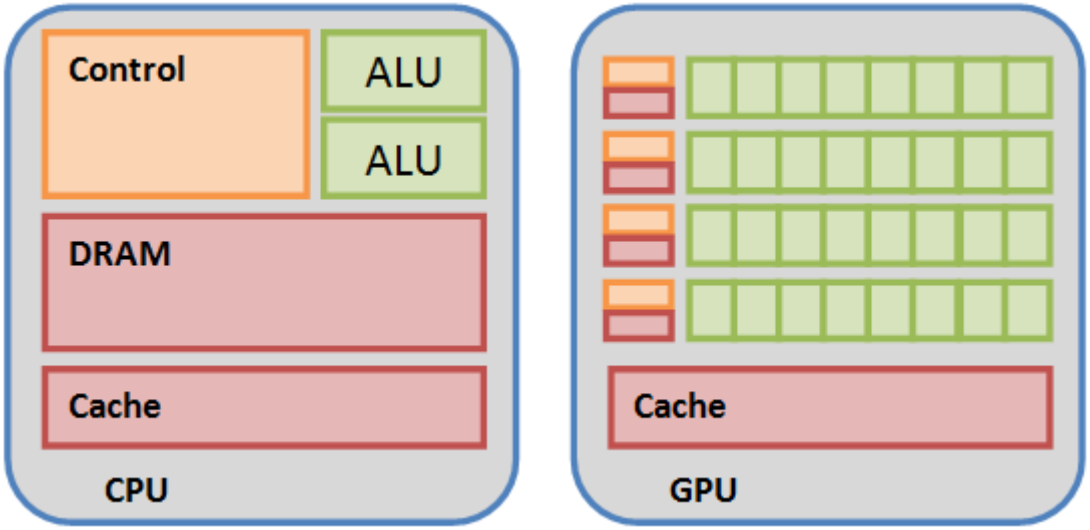
11 - Výsledek segmentace po sdružení atraktorů

3 Technologie CUDA

Nové grafické karty firmy NVIDIA umožňují na své architektuře provádět mimo běžné grafické úkony také obecné výpočty. Zkratka CUDA (Compute Unified Device Architecture) tedy znamená unifikovanou architekturu zařízení pro výpočty. Tato architektura nabízí softwarovým vývojářům snadný přístup k výpočtům pomocí již zavedených programovacích jazyků. Konkrétně se jedná o jazyk C, který je obohacen o funkčnost právě pro snadnou práci s grafickou kartou.

3.1 Popis

CUDA vývojáři pro využití nabízí především sofistikovaný paralelismus. Na grafických kartách se nachází mnoho samostatných výpočetních jednotek, které jsou schopny pracovat paralelně všechny najednou. Tyto jednotky nejsou tak komplexní jako na CPU, zato je jich však na kartě mnohonásobně více. Na následujícím obrázku 12 je vidět, že CPU věnuje paralelizaci zpracování a optimalizaci podstatnou část své struktury, zatímco GPU se soustředí především na maximalizaci počtu výpočetních jednotek.



12 - Porovnání CPU a GPU

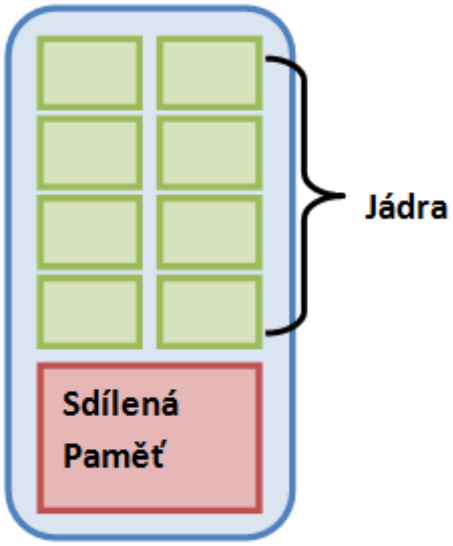
Na GPU se jako vývojáři musíme soustředit na adresovací problémy dat, které mohou být návrhem přepracovány na paralelní zpracování. V praxi se tato problematika dá snadno aplikovat na problémy při zpracování obrazu, videa a podobně. Každé výpočetní jednotce na kartě je například přiřazena malá část obrázku, kterou jednotka dostane ke zpracování. Toto mapování menších celků na mnoho paralelně pracujících jednotek však mohou využít mnohé programy řešící i zcela odlišné výpočetní problémy. Je třeba kvalitně analyzovat řešený problém a přepracovat jeho návrh právě pro využití tohoto paralelismu, který oproti sériovému zpracování nabízí výrazné navýšení dosažitelného výkonu. Ne všechny problémy jsou však paralelizovatelné natolik, aby efektivně využívaly výhod paralelního zpracování.

3.2 Využitelné aspekty

K využití nám technologie CUDA nabízí mnohé prostředky, které je nanejvýš vhodné při návrhu programu zohlednit. Veškeré tyto prostředky nabízí propracované funkce a výhody, díky kterým si můžeme práci nejen zjednodušit, ale mnohdy také zpřehlednit či lépe strukturovat. Využitím vhodných aspektů této technologie pro návrh námi řešeného problému však primárně získáme obrovskou možnost navýšení výpočetního výkonu aplikace jako celku.

3.2.1 Kernel

Oproti normálním funkcím v jazyce C je možno programovat na architektuře CUDA funkce nazývané kernely. Hlavní rozdíl spočívá v tom, že kernel je vykonáván najednou mnoha vlákny paralelně. Je dobré psát kernely co nejméně složité a snažit se funkcionalitu rozprostřít co nejvíce do šířky, což znamená mnoho na sobě nezávislých výpočtů, které se zpracovávají najednou. Kernel je vykonáván každým vláknem, pro které je aktuálně zavolán. Fyzicky jsou vlákna spjata s jednotlivými jádry v rámci daného multiprocesoru. Na obrázku 13 je znázorněna struktura jednoho multiprocesoru.



13 - Rozdělení multiprocesoru

Blok je logická jednotka, která v sobě obsahuje vlákna a je fyzicky zpracovávána multiprocesorem. Jeden blok je schopen virtuálně pojmout až 512 vláken, ačkoli fyzicky má multiprocesor pouze 8 jader. Vlákna se na něm totiž střídají ve vykonávání.

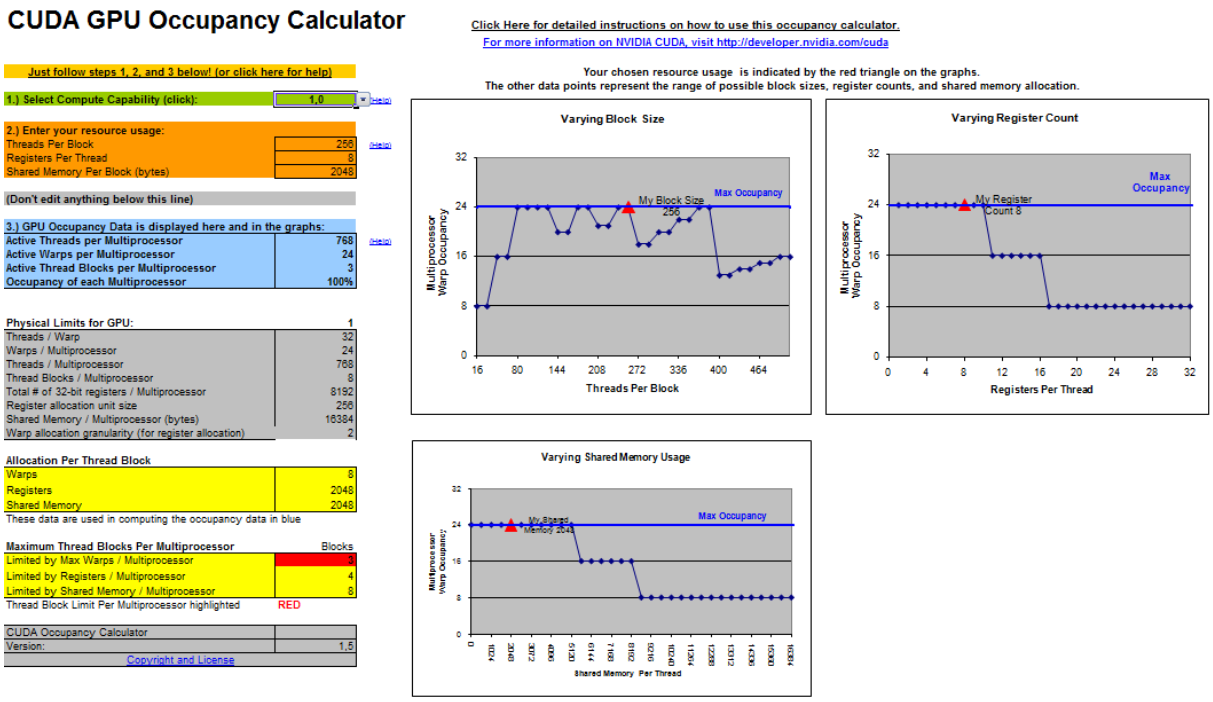
V rámci bloku se vlákna také mohou synchronizovat, což znamená, že se ve výkonu programu v daném okamžiku počká, až všechna vlákna dosáhnou daného synchronizačního bodu. Tato synchronizace je velice důležitá vzhledem k problémům, které mohou nastat při paralelním zpracování. Určitá vlákna mohou dospět k dané části kódu rychleji než ostatní a může například nastat situace, že ještě pro danou funkci nebudou připravena data, které mělo zajistit jiné vlákno. V takové situaci může reálně nastat to, že rychlejší vlákno sáhne na data, která mu nejsou určena, a dojde k selhání výpočtu, nebo špatným hodnotám ve výsledku. Dalším důvodem pro synchronizaci je fakt, že multiprocesor obsahuje pouze jednu instrukční jednotku. Pokud tak dojde k situaci, že mají některá vlákna jinou instrukci než zbytek vláken, je zpracování pomalejší, protože musí instrukční jednotka rozdělovat instrukce postupně. V případě, kdy jsou vlákna synchronizována, přiděluje instrukční jednotka danou instrukci všem vláknům najednou v jednom strojovém cyklu.

3.2.2 Globální paměť

Největší datový prostor pro ukládání na grafické kartě je označován jako globální paměť. Kapacita této paměti je obrovská, jelikož se většinou jedná o téměř veškerou RAM paměť dané karty, což dnes běžně bývá okolo 500 MB. Nejnovější grafické karty samozřejmě nabízejí o mnoho více. Před využitím této paměti je potřeba ji dynamicky alokovat a poté k ní přistupovat pomocí funkcí k tomu určených. Nahraná data jsou poté viditelná pro všechna vlákna. Hlavní nevýhodou této paměti je relativně nízká datová propustnost a fakt, že pro přístup k ní se musí fyzicky vyčkat mnoho strojových cyklů, které mohou být využity pro výpočty.

3.2.3 Registry

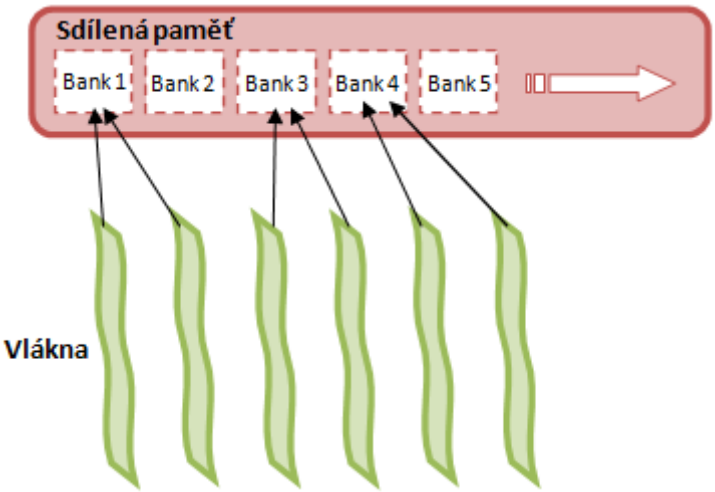
Každé vlákno má paměťovou část zvanou registry pro svou potřebu uložení lokálních proměnných a podobně. Tyto registry jsou fyzicky spjaty s jednotlivými jádry a dosahují proto výborné datové propustnosti a latence při přístupu k nim. Počet registrů je však pevně stanoven, tudíž je vhodné při složitějších kernelech hlídat si jejich obsazenost například počitadlem využití paměti, které je přikládáno k CUDA SDK. Celkový počet registrů se rozděluje zadanému počtu vláken v bloku. Je vhodné si poměr mezi počtem vláken a využitými registry hlídat. Při nízkém nebo nadměrném využití dostupných registrů pro jedno vlákno totiž program nedosáhne velké efektivity.



14 - CUDA Occupancy Calculator

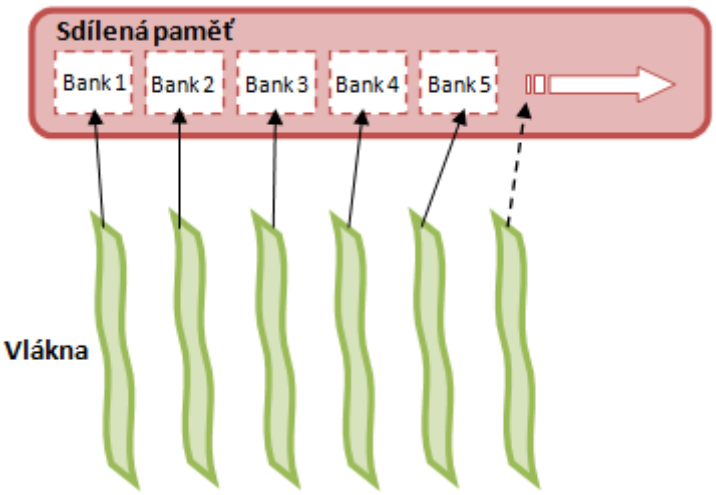
3.2.4 Sdílená paměť

Velice mocným nástrojem pro vývojáře je nabízená sdílená paměť. V rámci jednoho bloku k této paměti mohou přistupovat všechna vlákna v bloku obsažená. Hlavní výhodou této paměti je především velice nízká doba potřebná k přístupu, která dosahuje téměř hodnot registrů. Sdílená paměť je také svou kapacitou větší než výše zmíněné registry. Tato paměť je na současných kartách limitována na 16 kB, což pro potřeby mého programu bylo dostačující, ovšem pro mnoho jiných programů je tato kapacita malá. Nejdůležitější je především navrhnout program tak, aby vůbec tuto paměť efektivně využíval. Mnou navržený kernel je možno jednoduše upravit tak, aby využíval co největší část sdílené paměti.



15 - Nevhodný způsob přístupu do sdílené paměti

Sdílená paměť je dynamická a má tudíž tu vlastnost, že dochází k destrukci dat buňky při čtení i zápisu. Po každém přístupu je tak nutno data obnovit, což při vícenásobném přístupu k jedné buňce znamená zdržení. Při práci s touto pamětí je proto vhodné zvolit přístup k datům tak, aby více vláken nesahalo na stejnou adresu zároveň. V takovém případě by nastal konflikt a jedno vlákno by muselo čekat na druhé, až data načte, což by vedlo ke snížení propustnosti této paměti. Pro zamezení těchto konfliktů je nutno navrhnout přístup tak, aby do jedné buňky nepřistupovalo více než jedno vlákno.



16 - Správný způsob přístupu do sdílené paměti

3.3 Použité nástroje

K architektuře CUDA je možno použít několik existujících nástrojů. Některé z nich, jako CUDA Toolkit, je nutné mít pro správnou funkčnost. Jiné zase vývojáři nabízí automatizaci vybraných úkonů, které nejsou zrovna triviální a vyžadují čas a znalosti pro své řešení.

3.3.1 NVIDIA Parallel Nsight

Nejnovější technologie firmy NVIDIA nese označení Parallel Nsight (dále jen Nsight), což v překladu naznačuje možnost nahlédnout pod pokličku všeho paralelismu skrytého v grafické kartě. Se standardními nástroji Visual Studia 2008 sice můžeme vytvářet programy pro technologii CUDA, ale nemůžeme náš kód běžící na GPU nijak ovlivnit či sledovat. Právě toto si Nsight klade za cíl a díky novým nástrojům jako jsou NVIDIA Parallel Nsight Host (dále jen Host) a NVIDIA Parallel Nsight Monitor (dále jen Monitor) je toto vše nyní možné.

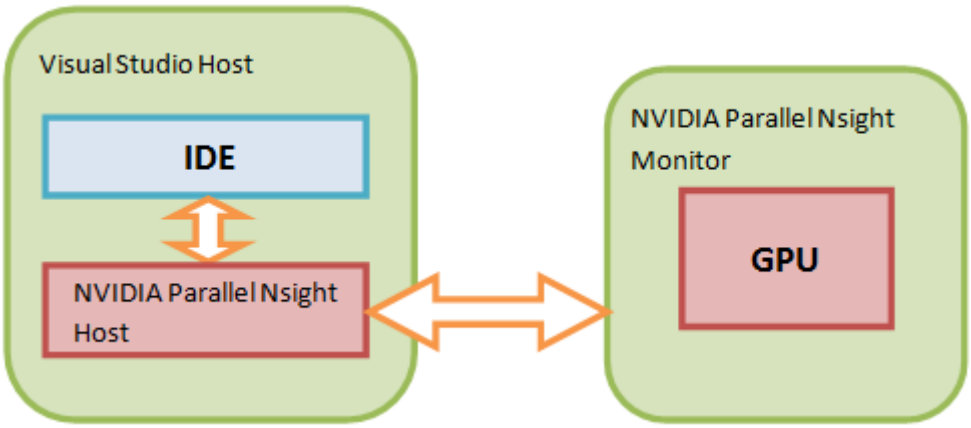
Nástroj Nsight v sobě zahrnuje vlastní kompilátor, který je upraven pro součinnost s ladicím režimem Visual Studia. V existujícím projektu je tedy nutné tento kompilátor nastavit jako výchozí. Dále je v projektu nutno nastavit sadu kompilačních pravidel pro Nsight, jelikož obsahuje svou vlastní. Veškeré kroky, jak nastavit projekt pro využití této technologie, jsou popsány v dokumentaci přiložené k instalaci Nsightu. Po úspěšném nastavení všech potřebných vlastností projektu můžeme využít funkcí této technologie.

Důležitým faktem této technologie je, že pro svou funkci vyžaduje dvě GPU. To znamená, že pro její využití nám nepostačí klasický stolní počítač s jednou grafickou kartou. Jediná možnost pro práci na jednom počítači je mít alespoň dvě GPU. Toto úskalí je dáno tím, že právě GPU, na kterém probíhá výpočet, je při ladicím režimu pozastaveno a tím pádem nemůže poskytovat odezvu na monitor. Nástroj Nsight se však skládá ze dvou hlavních částí, které nabízí další možnosti, jak se tomuto problému elegantně vyhnout.

Pro využití této technologie je potřeba již existující projekt nakonfigurovat. Postup pro nastavení projektu není triviální, ale je dobře popsán v dokumentaci, která se po instalaci tohoto balíku nástrojů stane jeho součástí. Tímto postupem jsem se řídil a projekt mi po správném nastavení s těmito nástroji spolupracoval.

3.3.1.1 NVIDIA Parallel Nsight Monitor

Srdcem nástroje Nsight je bezesporu právě aplikace Monitor. Ta prostřednictvím svého rozhraní poskytuje veškerou funkčnost této technologie, tudíž je nutno pro její využití mít ji správně nainstalovánu a nastavenu. Na obrázku 17 vidíme, jak Visual Studio s Monitorem pracuje.



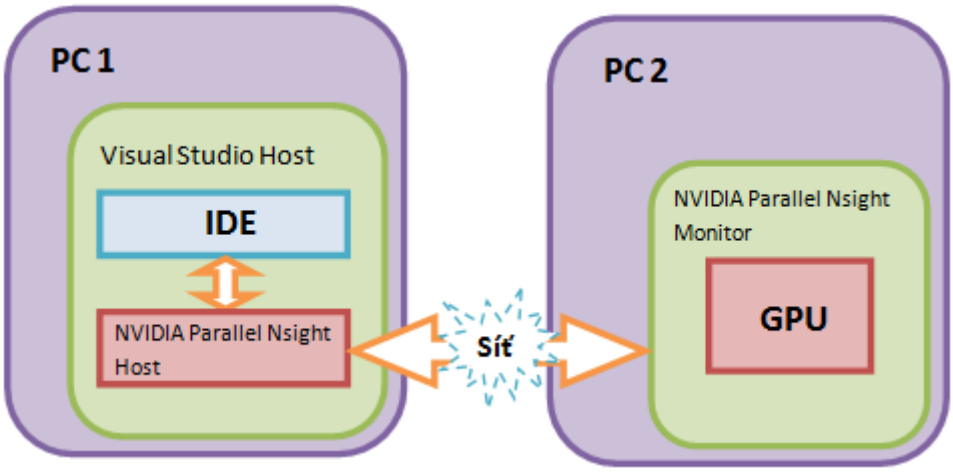
17 - Komunikace Visual Studia s NVIDIA Parallel Nsight Monitorem

Na jednom počítači by tedy situace vypadala následovně. Jedno GPU by bylo vyhrazeno pro běh grafického rozhraní systému a druhé by bylo ovládáno prostřednictvím Monitoru. V tomto případě si můžeme běžící kód na grafické kartě v libovolném bodě pozastavit a prohlédnout si vše, co potřebujeme. Tento přístup má obrovskou výhodu v tom, že nejsme nijak limitováni časem běhu našeho CUDA programu. Díky technologii TDR (Timeout Detection & Recovery) systému Windows jsme totiž dle standardního nastavení omezeni na pouhé 2 sekundy běhu kódu na GPU, než dojde k restartování jádra GPU, což má za následek krátkodobý výpadek obrazu (při použití jednoho GPU) a ztrátu veškerých dat na grafické kartě. Monitor je schopen TDR vypnout, což je nutno zvolit v jeho nastavení. Poté je možno využívat jeho funkce bez omezení.

3.3.1.2 Vzdálené ladění

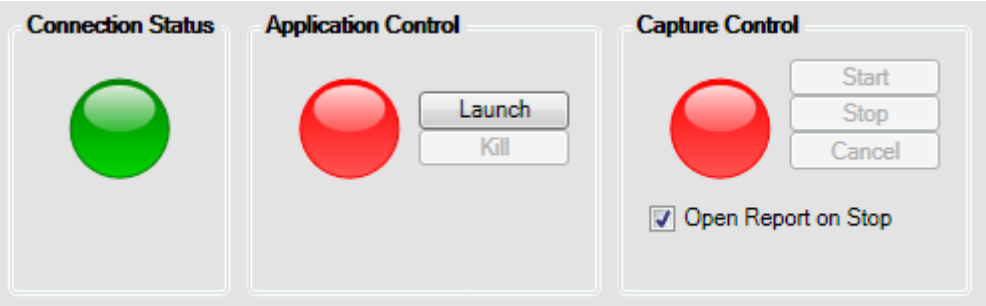
Monitor však nabízí i daleko zajímavější alternativu přístupu ke svým funkcím. Jedná se o vzdálený přístup k počítači, kde je Monitor spuštěn. Tento přístup je realizován prostřednictvím síťového spojení mezi počítači skrze protokol TCP/IP.

Vzdálený přístup k funkcím Monitoru tedy vyžaduje mít jej spuštěný na vzdáleném PC. Ve Visual Studiu nastavíme ve vlastnostech CUDA Debuggingu IP adresu daného vzdáleného stroje, ke kterému se chceme pro nabízené služby připojit.



18 - Schéma vzdáleného přístupu

Stav připojení můžeme zkontrolovat v záložce New Analysis Activity v hlavní nabídce Visual studia pod slovem Nexus. V příslušném okně (viz obrázek 19) vidíme stav spojení se vzdáleným Monitorem indikovaný barevnou ikonou. Další dvě ikony se týkají indikace, zda na Monitoru právě probíhá výpočet a v jakém je stavu. Pomocí přiložených tlačítek můžeme na dálku ovlivňovat běh daného programu.



19 - Indikace a ovládání aktuálního stavu

Takto můžeme využívat vzdálené PC s GPU umožňujícím použití technologie CUDA, aniž bychom na vlastním stroji takové GPU měli. V době běhu výpočtu na vzdáleném GPU je samozřejmě danému vzdálenému PC zamezeno v přístupu ke GPU, což má za následek pozastavení odezvy na jakékoli připojené zobrazovací zařízení. V současné době je technologie Nsight pouze ve verzi Beta a není veřejně přístupná.

Při vývoji svého programu jsem vzdáleného přístupu k Monitoru využil. Veškeré snímky z tohoto nástroje proto pochází ze zpracování na vzdáleném PC.

3.3.2 CUDA Visual Studio Wizard

Jelikož nastavení nového projektu, například obyčejné konzolové aplikace, tak, aby využíval technologii CUDA, zahrnuje řadu kroků, je dobré využít automatizovaný prostředek pro zakládání prázdných projektů. Pro tyto účely slouží volně dostupný průvodce tvorbou nového CUDA projektu, který je k nalezení na stránkách www.sourceforge.net. Po instalaci se tento průvodce začlení do výchozí nabídky při volbě nového projektu, odkud je možné jej využívat.

4 Algoritmus segmentace

Pro svou práci jsem od MRL (Media Research Lab v rámci FEI) obdržel již fungující algoritmus, který je schopen segmentovat obrázky dle zadaných parametrů. Daný algoritmus pro svou práci klasicky využívá CPU a je naprogramován v jazyce C.

4.1 Popis algoritmu pro CPU

Algoritmus pracuje s obrázky ve formátu RAW, kde jsou pro jeden pixel za sebou složky R, G a B. Všechna větší pole jsou v programu alokována v paměti RAM dynamicky. Po načtení obrázku do takto vyalokované paměti se stejně vytvoří pole pro uložení výsledků. Obrázek se do pole ukládá ve formě pole desetinných čísel, kde každá složka nabývá hodnoty od 0 do 1. Následně se hodnotami naplní pole, které určuje oblast zájmu pro výpočet segmentace. Oblast zájmu pro nás v obrázku může být výsek ohraničený minimálními a maximálními hodnotami souřadnic. Dále pak můžeme specifikovat rozsah pro každou barevnou složku obrázku, kterou chceme do výpočtu zahrnout.

Po zadání klíčových hodnot, což jsou šířka výpočetního okna a šířka barevného rozsahu, se spustí se všemi potřebnými parametry hlavní procedura MeanShift, která má na starosti kompletní průchod obrázku a určení všech segmentů.

Uvnitř procedury MeanShift se pomocí dvou vnořených cyklů prochází všechny relevantní pixely, které jsou určeny hranicemi v poli oblasti zájmu. Každý pixel si do pole načte své počáteční souřadnice a také příslušné barevné složky. Následuje další vnořený cyklus, který provádí iterace výpočtu do doby, než je ukončen maximálním počtem iterací, nebo podmínkou minimálního posunu. V jeho těle dochází k volání procedury GetNewPointPosition2dRGB, což je funkce, která se stará o výpočet posunu dané iterace.

Proceduře GetNewPointPosition2dRGB jsou předány potřebné parametry včetně celého obrázku. Po vytvoření proměnných pro uložení mezivýsledků se pomocí dvou vnořených cyklů prochází všemi pixely čtvercového výpočetního okna. Každý pixel je zkontrolován proti oblasti zájmu výpočtu, a pokud projde, je spočítána jeho vzdálenost vůči centrálnímu pixelu okna. Pomocí kernelu je ze vzdálenosti vypočtena váha daného pixelu. V dalším kroku jsou souřadnice a barevné složky vynásobeny touto váhou a přičteny do odpovídajících sum. Z těch se posléze vypočítá nová pozice pro posun pomocí dělení výsledné složky sumou vah. Důležitým výstupem této procedury je délka posunu iterace, která se počítá na závěr. Podle té procedura nad ní pozná, zda již má ukončit daný výpočet z důvodu příliš malého posunu.

Po dokončení cyklu s iteracemi se pro počítaný pixel musí zapsat, na jaké pozici skončil. Pro uchování těchto informací používám dvě jednorozměrná pole. Do prvního pole zapisuji na index výchozího pixelu číslo pixelu, ve kterém skončil. V druhém poli pro koncové pixely zaznamenávám počet pixelů, které do nich dospěly.

V této fázi máme ve výsledkových polích uložen výsledek segmentace. Ten můžeme reprezentovat například tak, že obrázek převedeme do černobílého spektra pomocí jednoduchého přepočtu v cyklu. Následně budeme procházet výsledková pole a kdykoli narazíme na pixel, do kterého doběhlo dostatek pixelů (určeno parametrem), obarvíme pixely patřící do tohoto segmentu stejnou barvou. Tímto postupem získáme v původním obrázku vybarvené jednotlivé nalezené segmenty.

4.2 Ukázka

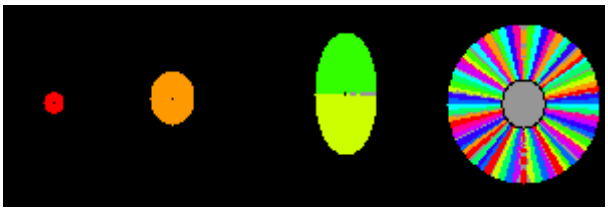
Pro ukázku jsem použil obrázek, který jsem vytvořil pro účely snadné demonstrace výsledků. Obsahuje čtyři různě velké kruhy tvořené šedými pixely, viz obrázek 20. Parametry algoritmu byly nastaveny tak, aby měl dostatečný počet iterací pro dosažení výsledku a aby nebral v potaz černé pixely pozadí. Pro ukázku jsem nastavil, aby z výsledných segmentů algoritmus vybíral pouze ty, které dosáhnou počtu 30 pixelů a více z důvodu přehlednosti výsledku.

20 - Původní obrázek

Na obrázku 21 je znázorněn výsledek segmentace s nastavenou velikostí poloviny hrany výpočetního okna na 10 pixelů. Jak je z obrázku patrné, došlo pouze k částečné segmentaci, což je dáno malým výpočetním oknem, které nebylo schopno zanořit se dostatečně hluboko do šedých kruhů. Jakmile se totiž okno ocitne v situaci, kdy má všechny pixely podobné či stejné, vypočtou se nové souřadnice pro přesun iterace poblíž středu aktuálního okna, což vede k ukončení výpočtu. Algoritmu se tedy podařilo správně určit pouze nejmenší kruh, který vybarvil červeně. V druhém kruhu je vidět snaha algoritmu dostat se do jeho středu, což dokazují segmenty s atraktory poblíž středu.

21 - Výsledek segmentace 1

Obrázek 22 ukazuje výsledek segmentace s velikostí poloviny hrany výpočetního okna 30 pixelů. Algoritmus byl schopen správně rozpoznat první dva kruhy z levé strany. Třetí kruh z leva je rozdělen na dva segmenty z důvodu zaokrouhlení souřadnic a pozastavení výpočtu kvůli malým posunům, což vedlo k určení atraktorů vedle sebe namísto jednoho jediného atraktoru uprostřed. Poslední největší kruh obsahuje mnoho malých segmentů, ze kterých je patrná velikost poloviny hrany výpočetního okna. Situaci s atraktory určenými u sebe by vyřešilo sdružování atraktorů, které bylo popsáno v jedné z předchozích kapitol.



22 - Výsledek segmentace 2

4.3 Popis vlastností a efektivity

Hlavní procedura segmentace MeanShift v sobě obsahuje tři do sebe vnořené cykly, pomocí kterých prochází souřadnice a iterace. Uvnitř posledního cyklu se volá procedura GetNewPointPosition2dRGB. Ta ve svém těle obsahuje další dva vnořené cykly, kterými prochází pixely výpočetního okna. Celý algoritmus posunů má tedy v sobě dohromady pět úrovní vnořených cyklů. Vzhledem k množství operací, které jsou v každé části výpočtu obsaženy, je tato struktura programu velice náročná na systémové prostředky, a proto jí vcelku dlouho trvá i výpočet pro ne příliš velké obrázky.

4.4 Paralelizace pomocí OpenMP

Většina aktuálně používaných procesorů ve stolních počítačích již obsahuje dvě jádra i více. Výše zmíněný algoritmus bez problému zabere pro svůj výpočet celých 100% výkonu jednoho jádra. Pomocí API OpenMP je možno v jazyce C jednoduše využít paralelního zpracování na CPU. Pro funkčnost tohoto nástroje je nutno použít knihovnu omp.h, kterou vložíme do našeho programu. Následující ukázka kódu 1 demonstruje využití paralelizace touto knihovnou pro urychlení hlavních dvou cyklů procedury MeanShift.

```
#pragma omp parallel for schedule (static,1)
  for   (int y = round(roi[1][0]); y <= round(roi[1][1]); y++ ){
    for (int x = round(roi[0][0]); x <= round(roi[0][1]); x++ ){
```

Ukázka kódu 1 – Použití OpenMP

Takto upravený algoritmus využívá pro svou funkci obě jádra mého procesoru zároveň a je schopen obě bez problému zatížit na 100%. Při tomto nastavení program automaticky využije co nejvíce dostupných jader. Program potom dosahuje znatelně kratší doby běhu. Je tedy vidět, že použití této knihovny je relativně jednoduché a nabízí velký prostor pro vylepšení stávajících programů běžících na CPU.

5 Adaptace algoritmu segmentace na CUDA

Na popsaném algoritmu pro CPU je zřetelně viditelné, že adresování dat vyžaduje pro svůj běh obrovskou režii, která velkou měrou snižuje výkon celého programu. V následujících kapitolách bude popsáno, jak se co nejlépe vypořádat s novým návrhem tak, aby zohledňoval co nejvíce možných výhod, které nám technologie CUDA nabízí.

5.1 Hlavní myšlenky

Při analýze problému pro vývojové prostředí s nasazením rozsáhlé paralelizace, je nutno přemýšlet jiným způsobem, než jsme zvyklí u klasického programování pro běžné procesory. Všechno, co tak dobře známe, má zde rozdílné využití oproti sériovému způsobu zpracování. Vše, co bychom v klasickém programu chtěli řešit vnořenými cykly, musíme nejlépe navrhnout tak, aby se program nerozrůstal do hloubky, jak je tomu právě u cyklů, ale do šířky. Toto základní rozeznání možných paralelních struktur v již existujícím programu je klíčové k dosažení co nejlepších výsledků. Problematika paralelizace je velice rozsáhlá a vyžaduje alespoň základní pochopení funkčnosti dané architektury.

Můj návrh nového algoritmu pro technologii CUDA je postaven primárně na rolích třech základních prvků paralelismu CUDA. Konkrétně se jedná o využití abstrakce gridu, bloku a vlákna pro namapování částí řešeného problému na tyto jednotky. Každá z těchto jednotek má navíc fyzicky stanovenu horní hranici, která omezuje maximální počet využití dané jednotky.

- Grid – může v sobě obsahovat maximálně 65535×65535×1 bloků
- Blok – je schopen virtuálně pojmout nanejvýš 512 vláken
- Vlákno

Grid, jakožto nejrozsáhlejší abstrakce, představuje v mé koncepci celý obrázek, kdy jednotka tohoto gridu je jeden pixel se všemi svými složkami. Maximální velikost zpracovávaného obrázku je tedy 65535×65535 pixelů, což dělá 4,294 GPix. Takový rozsah stačí pro zpracování běžného obrázku.

Blok má v mém návrhu roli středového pixelu výpočetního okna, pro který výpočet začne. V rámci bloku se tudíž bude počítat celá cesta pixelu od jeho počátku až do konce jeho posunů. Konstrukce bloku v sobě zahrnuje také sdílenou paměť, kterou blok využívá mimo jiné pro ukládání dat o daném pixelu. K takto uloženým informacím totiž budou poté moct přistupovat jakékoli vlákna bloku, což je jedna z nejdůležitějších vlastností celého návrhu.

Vlákno, jako poslední a nejmenší stavební celek, sehrává v celém navrženém systému hlavní výkonovou jednotku, pro kterou programujeme kernel. Mnou použitá abstrakce dvourozměrného pole vláken slouží k namapování jednotlivých pixelů výpočetního okna na odpovídající vlákna.

5.2 Rozbor částí kódu

Kód mého programu se drží postupu, který jsem popsal u programu pro CPU. Důležité části, které se při využití paralelismu změnily, jsou popsány v následujících kapitolách.

5.2.1 Paralelizace smyček

```
vazene_x[IDvlakna] = index[0]*vahy[IDvlakna];
vazene_y[IDvlakna] = index[1]*vahy[IDvlakna];
vazene_r[IDvlakna] = buffer[0]*vahy[IDvlakna];
```

Ukázka kódu 2 – Paralelizace smyček

Jednou z nejvíce důležitých vlastností mého programu je předělání cyklů z původního algoritmu pro CPU na paralelní zpracování pomocí vláken. Klíčová myšlenka je v definování správného počtu vláken. Z výše uvedeného vysvětlení tedy vyplývá, že budeme-li například chtít zpracovávat čtvercové výpočetní okno s polovinou velikosti hrany 10 pixelů, budeme potřebovat 441 vláken, což je druhá mocnina hrany o velikosti 21 pixelů (10+10+1 středový).

Každé vlákno bloku provede podle svého indexu konkrétní výpočet, který by jinak musel být prováděn postupně právě cyklem. Tím jsme přenesli strukturu výpočtu ze smyčky do šířky a jsme tak schopni během jednoho strojového cyklu zpracovat x-krát více výpočtů, než na CPU, což se zajisté pozitivně promítne do rychlosti zpracování.

5.2.2 Využití sdílené paměti

Sdílená paměť bloku je v mém programu využita pro sdílení důležitých informací o postupu výpočtu, kterými se řídí určité kroky algoritmu, a pro ukládání mezivýsledků z paralelního zpracování.

```
__shared__ float pos[5];
__shared__ int maxi;
__shared__ bool b;
```

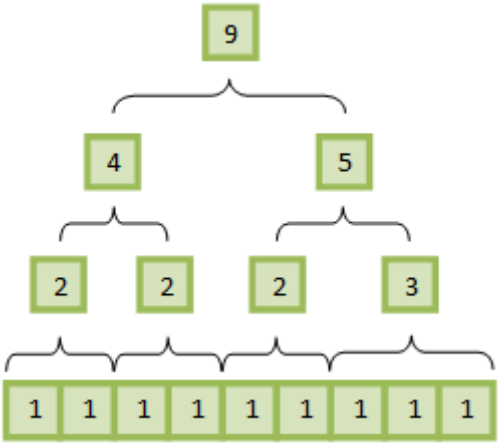
Ukázka kódu 3 – Definování proměnných ve sdílené paměti

Nejmohutnější pole, konkrétně váhy a vážené parametry počítaného pixelu, slouží k uchování dat z paralelního zpracování nahrazujícího smyčky. Tato data musí být ve sdílené paměti proto, aby k nim mohlo přistoupit jakékoli vlákno bloku. Zároveň se jedná o vcelku objemná data, která by se do registrů nevešla. Zápis hodnot do těchto polí navíc bude dosahovat dobré datové propustnosti. Velice rychle tak budeme mít paralelně vypočteny vážené parametry daného výpočetního okna, které budou čekat připraveny ve sdílené paměti na sečtení.

Další sdílené proměnné se používají pro součet jednotlivých vážených položek, pro uchování pozice středu, barevných složek a také se jimi řídí chod paralelního sčítání. Tato metoda paralelního součtu je popsána v následující kapitole.

5.2.3 Paralelní součty

Připravená pole s paralelně vypočtenými váženými parametry je potřeba sečíst do pole sum ve sdílené paměti. Pro každé pole by standardně bylo nutno cyklem provést stovky operací sečtení, což by v již dobře paralelně fungujícím algoritmu znamenalo markantní pokles rychlosti výpočtu. Další důvod pro aplikaci paralelního součtu je ten, že v CUDA nefunguje postup, kdy bychom nechali vlákna inkrementovat danou položku pole sum odpovídající hodnotou váženého parametru.



23 - Ukázka principu paralelního součtu

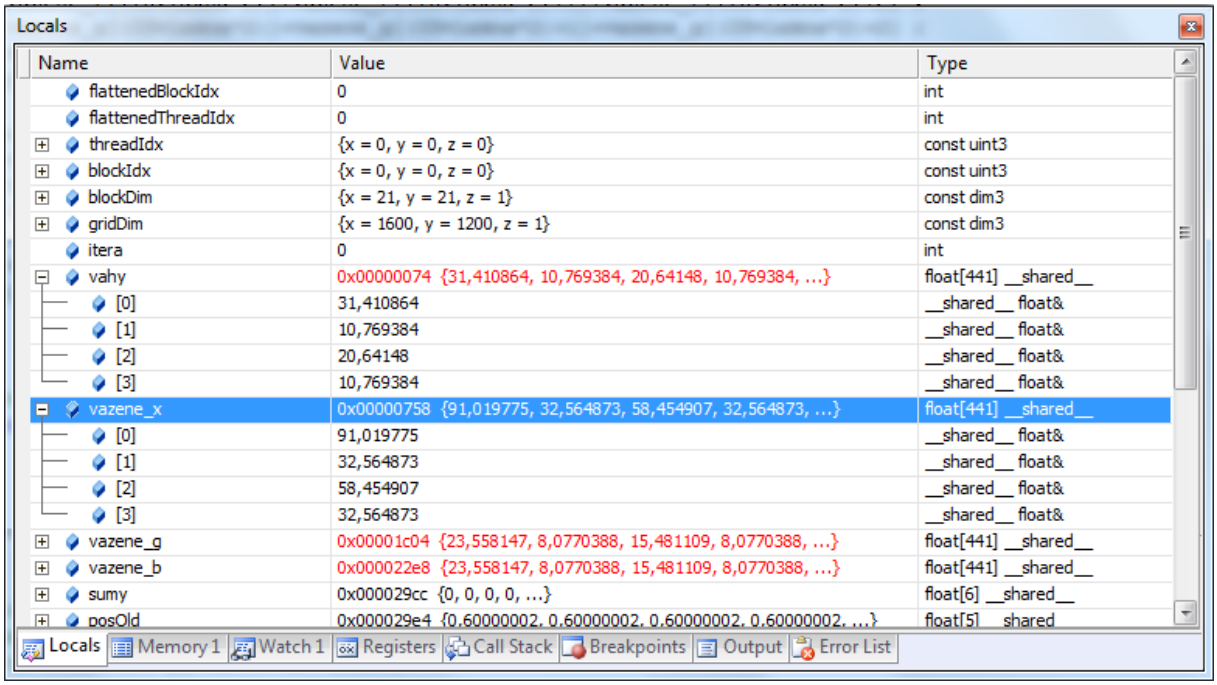
Mnohem sofistikovanější přístup představuje paralelní součet, který je demonstrován na přiloženém obrázku 23, který ukazuje, jak lze ve třech krocích sečíst devět sčítanců. Každé vlákno vyhodnotí dle parametrů, které jsou uloženy ve sdílené paměti, jakou roli aktuálně v součtu sehrává a podle té vykoná jeden ze tří postupů.

- Pokud je vlákno standardně nad dvěma nesečtenými buňkami aktuálně sčítaného pole, provede jejich součet.
- Pokud je vlákno na konci aktuálně sčítaného pole a zároveň je počet sčítanců lichý, provede sečtení posledních tří buněk.
- Pokud je vlákno za rozsahem aktuálně sčítaného pole, počká v nečinnosti na dokončení celého paralelního součtu.

Touto paralelní metodou sčítání pole jsme schopni značně urychlit zpracování všech součtů vážených parametrů. Vzhledem k vysokému počtu ušetřených strojových cyklů pro dokončení klasického sčítání pomocí cyklů při takto hustě obsazených polích se zkrácení doby běhu celého algoritmu určitě projeví.

5.3 Konkrétní ukázka paměti z remote debuggingu

Následující obrázek 24 byl pořízen při vzdáleném přístupu k NVIDIA Parallel Nsight Monitoru a ukazuje obsah používaných proměnných a konkrétní výpis prvních pěti prvků pole ze sdílené paměti.



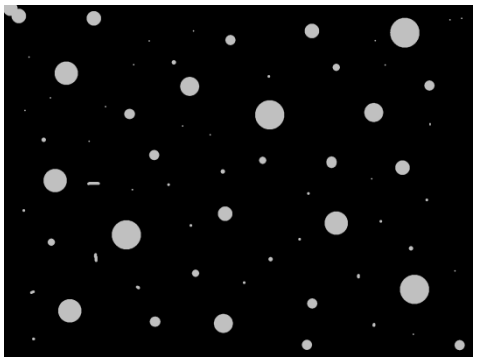
24 - Rozpis proměnných při remote debuggingu

6 Dosažený výkon

Tato kapitola se bude zabývat konkrétními výsledky, které jsem naměřil u daných programů. Dosažené výsledky programu pro CPU jsem naměřil na svém stolním počítači, který je vybaven procesorem Intel Pentium D945, který obsahuje dvě jádra taktovaná na frekvenci 3,4 GHz. Nejlepší uvedené výsledky programu na technologii CUDA byly pořízeny na grafické kartě NVIDIA GeForce GTX 260, která se nachází ve vzdáleném počítači, jinak jsem program vyvíjel, kompiloval a ladil na své grafické kartě NVIDIA GeForce 8800 GT.

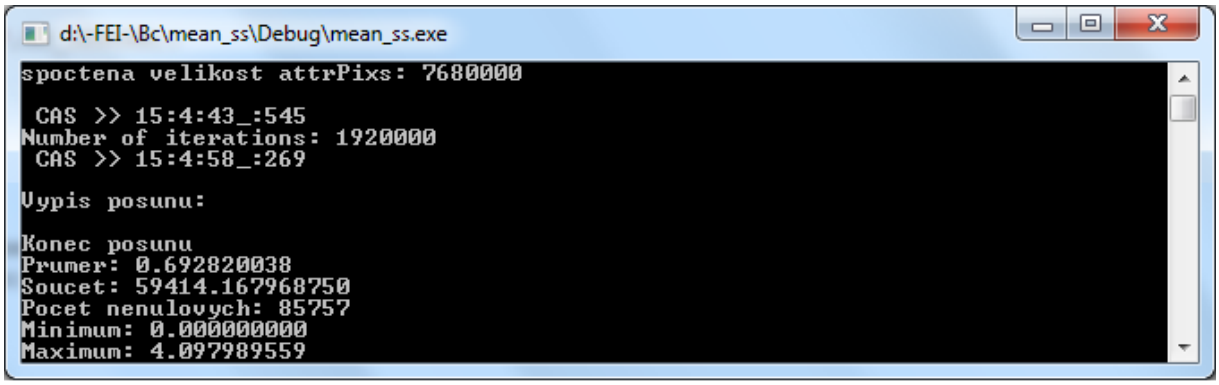
6.1 Shrnutí poznatků o CPU

Referenční program, který jsem obdržel, pracuje s CPU. Testování jsem prováděl na obrázku o velikosti 1600×1200 pixelů, který jsem pro testovací účely vytvořil. V obrázku jsou pouze šedé objekty na černém pozadí pro snadnou kontrolu výsledků obou programů. Pixelů, které splní podmínku pro oblast zájmu, co se barev týče, je v obrázku procentuálně relativně málo, takže nebudou výpočetní jednotky tolik vytíženy. I na tomto množství se však plně projeví výhody či neduhy jednotlivých programů.



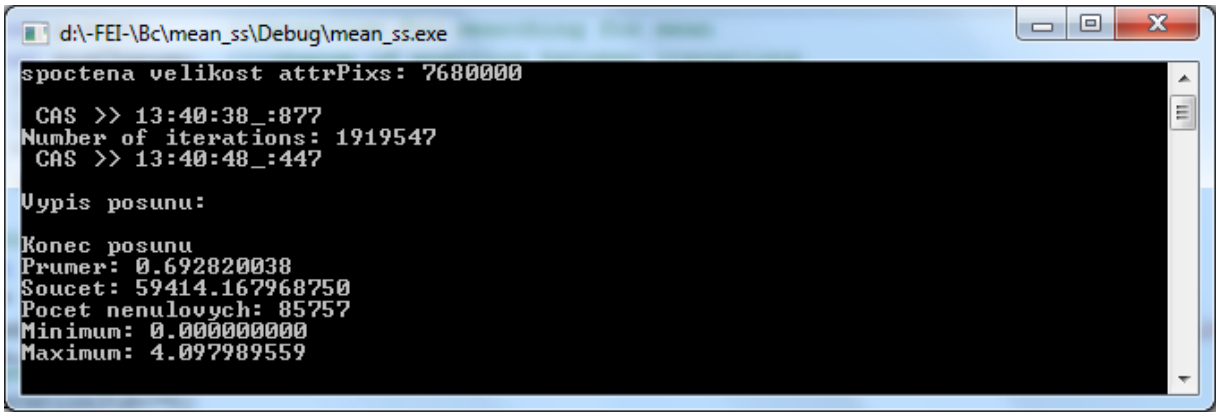
25 - Testovací obrázek

Program byl nastaven pro velikost poloviny hrany výpočetního okna 10 pixelů, což ve výsledku znamená procházení okna o hraně měřící 21 pixelů. Oblast zájmu byla nastavena souřadnicově na celou plochu obrázku a nastavení barevných složek neumožňovalo zahrnutí černých pixelů pozadí do výpočtu. Maximální barevný rozdíl byl nastaven na hodnotu 0,2. V následujícím obrázku 26 je zachycen výsledek běhu programu využívající standardně jedno jádro CPU.



26 - Výsledek CPU programu 1

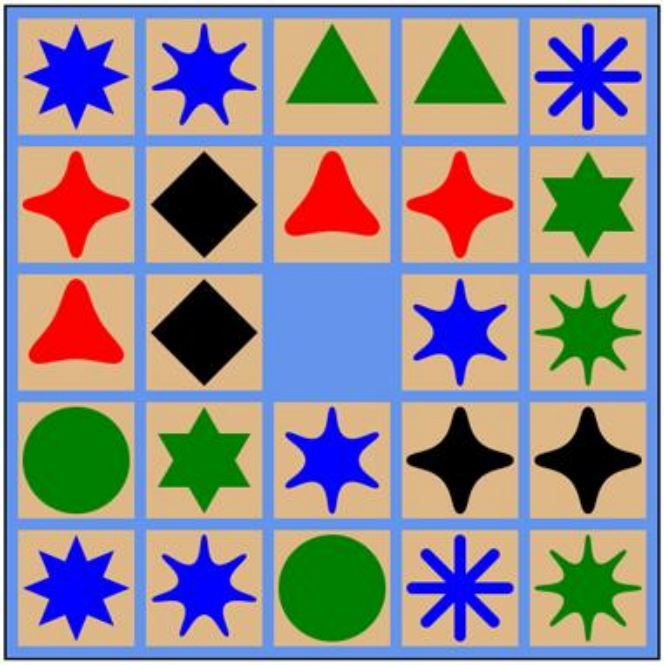
Program zadaný úkol splnil s výsledkem, který pro tento obrázek budu považovat za referenční, po uplynutí doby 14 sekund a 724 milisekund. V následujícím obrázku 27 program k výpočtu používal obě jádra CPU, která zatížil na 100%.



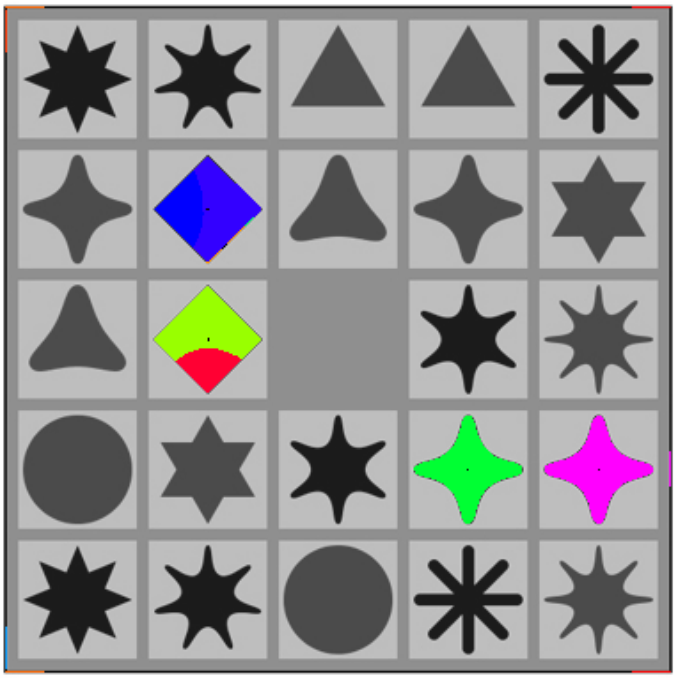
27 - Výsledek CPU programu 2

Výsledný čas doby běhu programu je v tomto případě 9 sekund a 570 milisekund.

Pro ukázku reálné segmentace pomocí tohoto algoritmu jsem vybral následující obrázek 28 o velikosti 382×382 pixelů, který obsahuje barevné tvary. Řekněme, že budeme chtít v obrázku určit černé segmenty. Po nastavení parametrů pro vyhledávání černých vzorků a dostatečnou velikost výpočetního okna (pro přehledný výsledek) program určí segmenty tak, jak je vidět na přiloženém obrázku 29.



28 - Příklad reálného obrázku



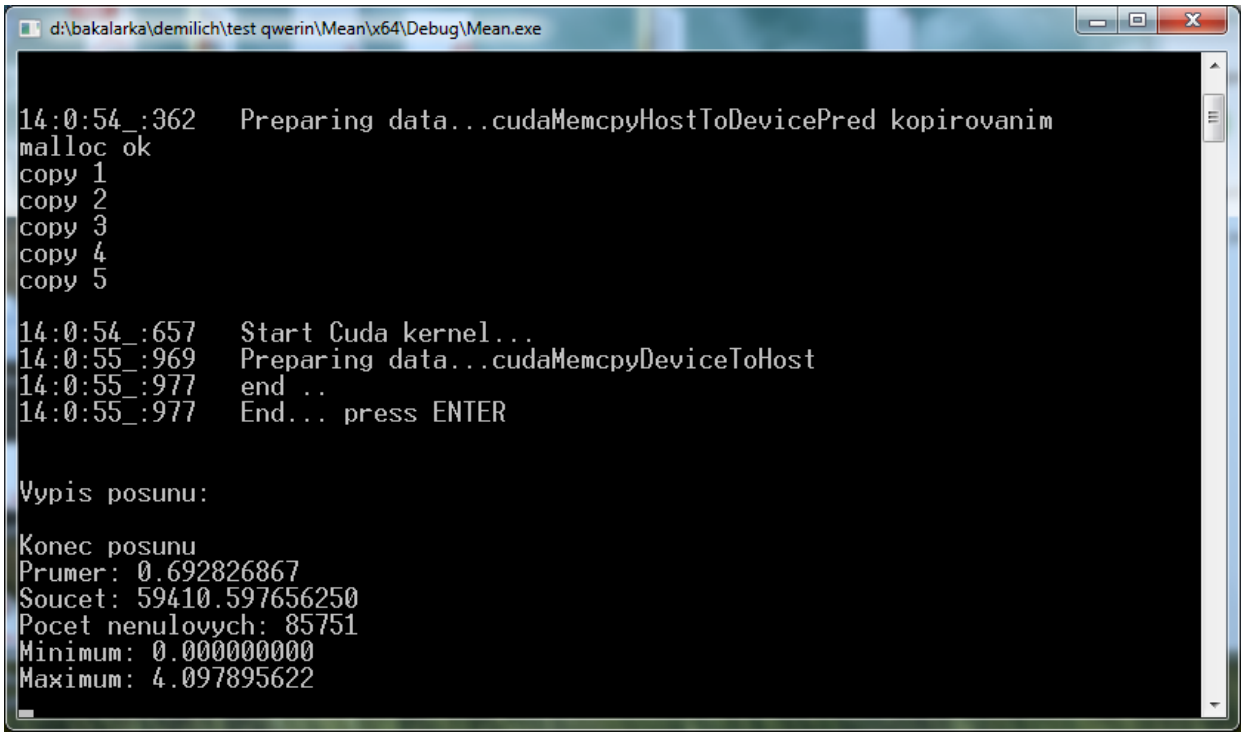
29 - Výsledek segmentace reálného obrázku

Program správně provedl segmentaci a po sdružení atraktorů bychom měli určeny všechny zamýšlené segmenty. Z tohoto výsledku je patrné, že algoritmus mean shift je schopen dobře segmentovat obrázky a je proto v praxi využitelný.

6.2 Výsledky CUDA algoritmu

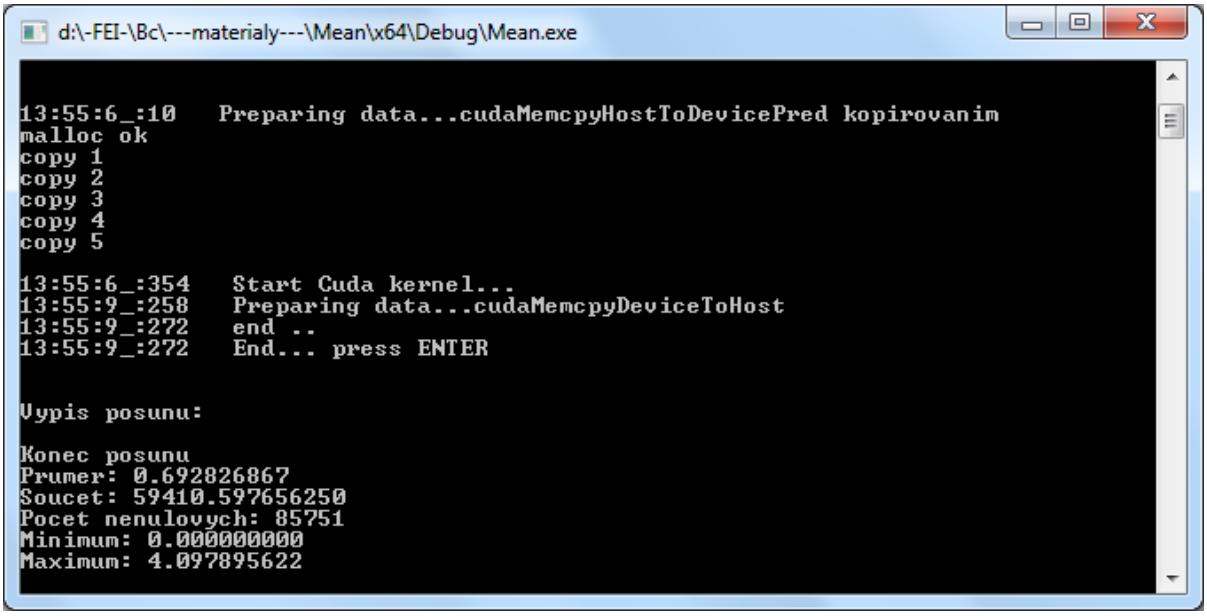
Mnou navržený a implementovaný program využívající paralelního zpracování jsem také testoval na výše uvedeném obrázku. Parametry výpočtu byly nastaveny shodně s programem pro CPU. Výpočetní okno mělo velikost hrany 21 pixelů, což určuje 441 vláken do jednoho bloku. Velikost celého gridu byla dána velikostí obrázku, tedy 1600×1200.

Můj kernel byl schopen provést správně výpočet jedné iterace pro výpočetní okno s hranou měřící 21 pixelů za 1 sekundu a 312 milisekund, což dokazuje následující snímek 30 z běhu programu. Vypsané časy byly měřeny programově na straně vzdáleného počítače disponujícího grafickou kartou GeForce GTX 260.



30 - Výsledek CUDA programu 1

Pro porovnání jsem nechal tento program proběhnout i na své grafické kartě GeForce 8800 GT, která má fyzicky přesně polovinu multiprocesorů oproti kartě GeForce GTX 260. Výsledná doba běhu kernelu byla přesně 2 sekundy a 905 milisekund, což dokládá přiložený obrázek 31.



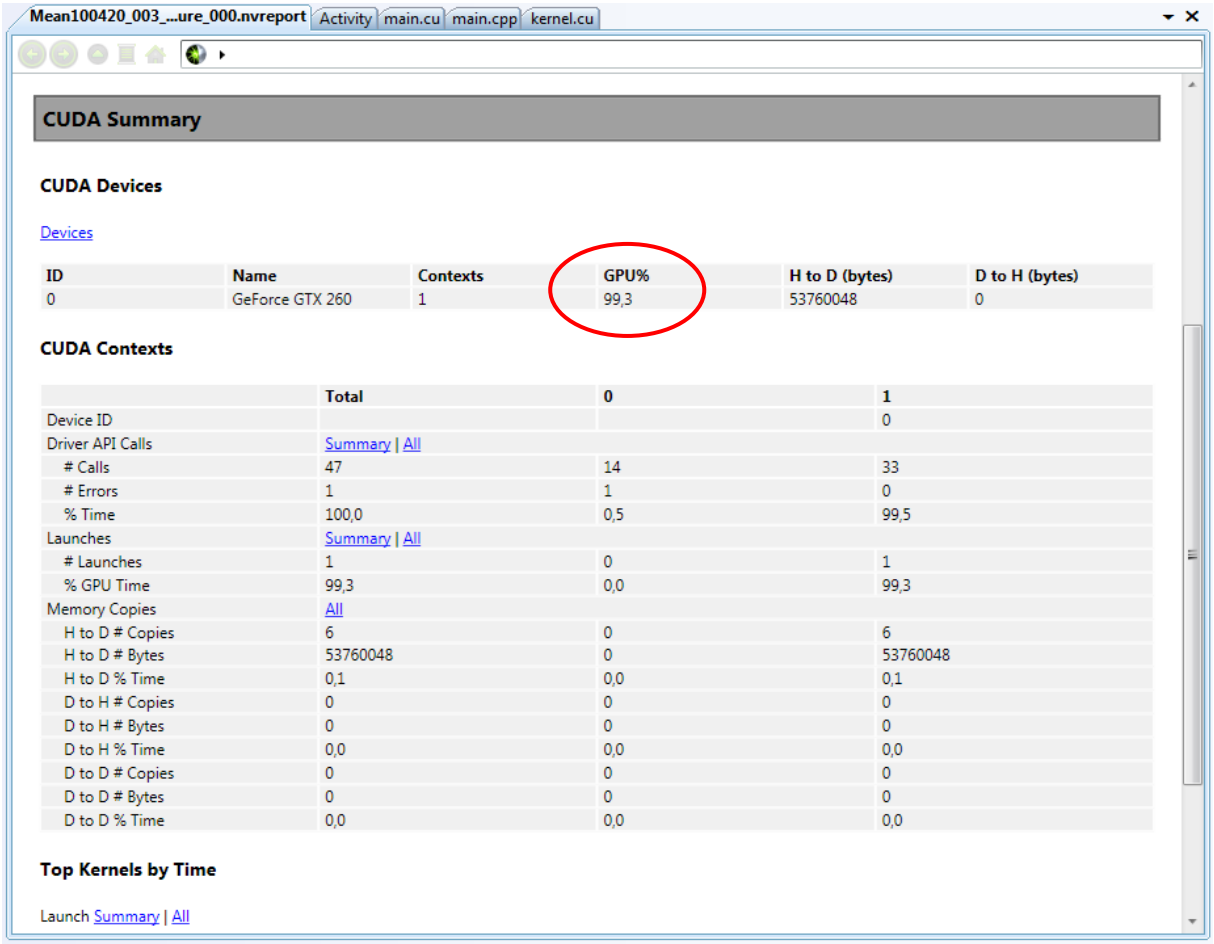
31 - Výsledek CUDA programu 2

Tento čas je o něco málo větší než dvojnásobek doby, kterou potřebovala karta s dvojnásobným počtem multiprocesorů. Na tomto výsledku je vidět dobrá škálovatelnost technologie CUDA oproti procesorovému řešení, kterému druhé jádro nepřineslo tak vysoký užitek. Na grafických kartách s větším počtem multiprocesorů by výpočet trval znatelně kratší dobu. Můj algoritmus využívá jeden

multiprocesor pro výpočet jednoho pixelu obrázku, což znamená, že je schopen využít například 10^6 multiprocesorů pro obrázek o velikosti 1000×1000 pixelů. Na stanicích obsahujících více samostatných grafických karet s podporou technologie CUDA by proto bylo možno výpočet dále rozdělit na dostupné multiprocesory a dobu běhu tak znatelně zkrátit.

6.2.1 Profil CUDA programu v NVIDIA Parallel Nsight Host

Pro svůj program jsem využil možnosti profilace v programu NVIDIA Parallel Nsight Host, který nabízí automatické vyhodnocování běhu programu. Ve verzi, se kterou jsem pracoval, bylo možno si pro jedno spuštění CUDA programu nechat vypsat až 4 hardwarová počitadla, která byla schopna měřit zadané parametry. Dozvěděl jsem se tak například, že můj CUDA program je schopen využít téměř veškerý výkon GPU (zakroužkováno v obrázku 32), že správně přistupuje k paměti při čtení i zápisu a kolik konkrétně využívá jaké paměti. Následující obrázek 32 ukazuje jeden z možných výpisů profilace programu.



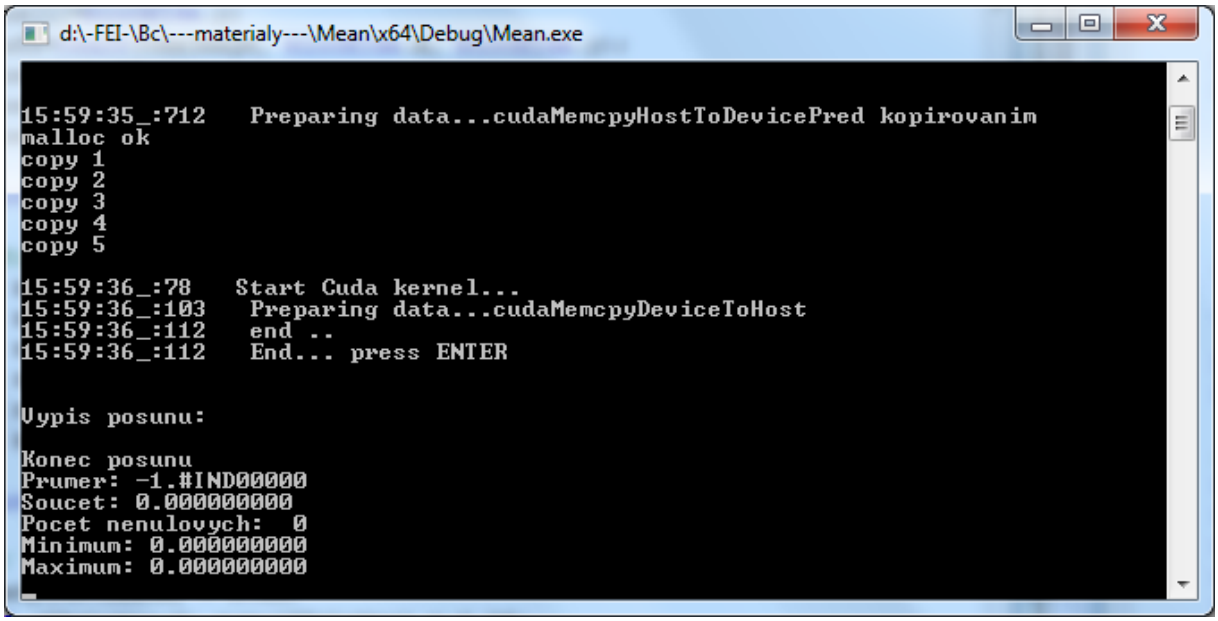
32 - Ukázka profilace CUDA programu

6.2.2 Problémy

Při vývoji svého CUDA algoritmu jsem nejdelší čas věnoval snaze o zprovoznění výpočtu pro více iterací. I při použití moderních ladicích programů jako je NVIDIA Parallel Nsight Host se mi nepodařilo objasnit příčinu toho, proč jednoduchým cyklem nelze provést další iteraci výpočtu. Pokud v cyklu, který se má starat o obsluhu jednotlivých iterací, zadám provedení jedné iterace, program bez problému korektně doběhne do konce se správným výsledkem. Jakmile však zvýším počet cyklů na 2

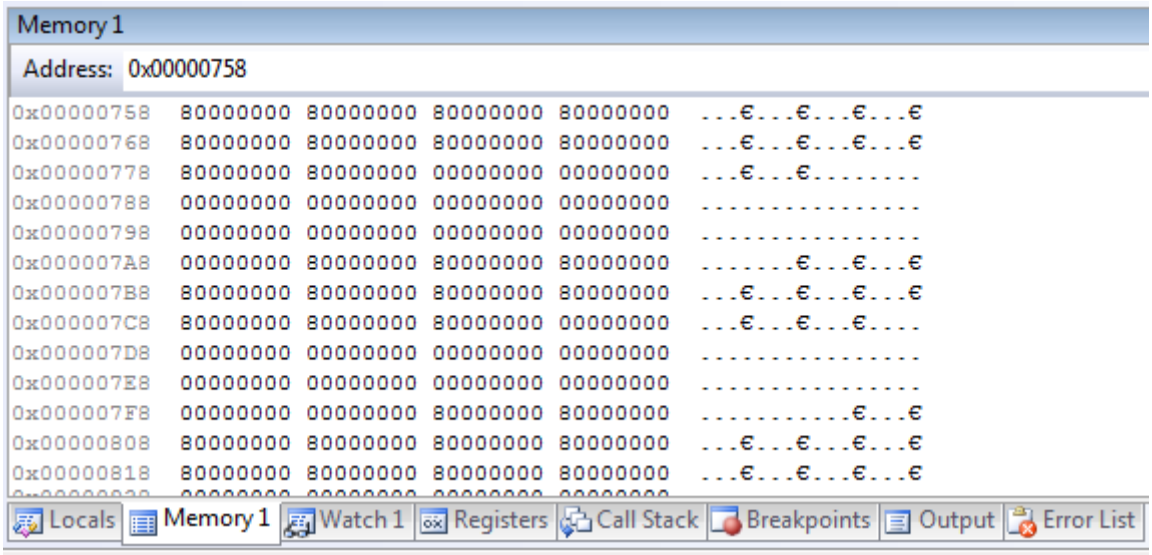
a více, dojde v programu k selhání, ačkoli jsou na začátku druhé iterace všechny proměnné správně nastaveny a všechna vlákna výpočtu aktivní. Při postupném sledování běhu programu jsem objevil dva chybové stavy, které nedovedu objasnit.

Následující obrázek 33 zachycuje běh programu na mém počítači s nastavenými dvěma iteracemi. Ačkoli programu trvá správné spočtení jedné iterace (pokud je zadána) více než 50 milisekund, je v tomto případě schopen za 25 milisekund doběhnout do konce s nulovým výsledkem. Pokoušel jsem se různými způsoby odchytit možnou chybu, ale program žádnou nenahlásí, což například v jiném případě, kdy záměrně zadám špatně počet vláken a podobně, udělá.



33 - Chybový stav 1 - nulové výsledky

Druhý zmíněný chybový stav jsem zkoumal za pomocí vzdáleného připojení k NVIDIA Parallel Nsight Monitoru. V ladicím režimu jsem postupně kontroloval každý krok výpočtu, až jsem narazil na chybu, která zjevně způsobuje nekorektní průběh algoritmu a je zodpovědná za špatný výsledek. V části sdílené paměti, například pro vážený parametr jedné souřadnice (viz obrázek 34), se objeví nejasné hodnoty i přesto, že bylo pole předtím vynulováno. V segmentu kódu, kde k tomuto jevu dochází, jsou aktivní všechna vlákna a situace před vynulováním je stejná jako pro jednu iteraci, ve které vše proběhne správně včetně synchronizace. Tyto hodnoty jsou dále v programu paralelně sečteny a zpracovávány, což vede ke zcela zkreslenému výsledku.



34 - Chybový stav 2 - výpis paměti

Řešení výše uvedených problémů jsem věnoval mnoho desítek hodin, než jsem odhalil alespoň místo výskytu této chyby. Je velice obtížné zpětně analyzovat program, který se na lokálním počítači chová jinak než při běhu na vzdáleném počítači. Tento fakt je také dán rozdílnými kompilátory pro překlad kódu pro klasické využití CUDA a NVIDIA Parallel Nsightu. Zatímco na mém stolním počítači mi program skončí chybovým stavem s nulami ve výsledku, na vzdáleném počítači se díky druhé popsané chybě dostane ke zcela scestným hodnotám.

6.2.2.1 Nepřesnost výpočtů

Při porovnání obrázků zachycujících běh obou programů je poznat určitý rozdíl ve vypočtených hodnotách. Tento rozdíl je způsoben přesností, se kterou architektura CUDA pracuje. Zda karta umí pracovat s jednoduchou, nebo dvojitou přesností se řídí číslem udávajícím výpočetní schopnost (Computing Capability) grafické karty. Moje grafická karta má výpočetní schopnost 1.0, jelikož se jedná o jeden z prvních typů, který vůbec technologii CUDA podporoval. Tento nejnižší stupeň je schopen počítat pouze s desetinnými čísly typu float, která jsou tvořena čtyřmi bajty dat. Z toho důvodu nemůže GPU počítat na stejný počet desetinných míst jako procesor, který má vnitřně reprezentaci datového typu float rozšířenu právě pro větší přesnost. Dvojitou přesnost počítání s desetinnými čísly nabízí až výpočetní schopnost 1.3, kterou mají pouze nejnovější grafické karty. V porovnání s procesorem je však i tato hodnota nižší. Tento fakt může pro určité programy vyžadující vysokou přesnost znamenat velké úskalí.

V mém algoritmu segmentace se ovšem nejedná o tolik závažný problém. Důležité je hlavně vypočtení nových souřadnic pro posun, což tímto problémem tolik ovlivněno není. Jedinou menší nevýhodou z tohoto problému plynoucí je, že se iterace ve výsledku posouvá o maličko kratší vzdálenost.

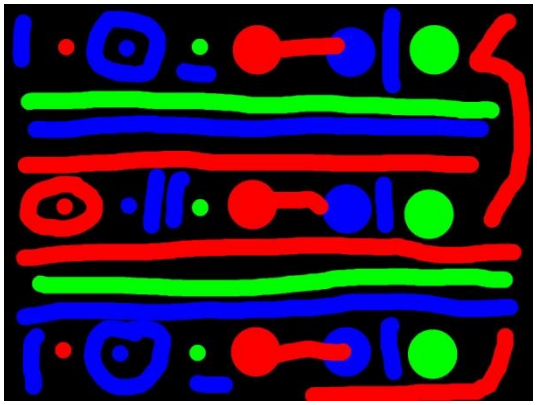
6.2.3 Prostor pro zlepšení a vývoj

Algoritmus, který jsem implementoval, je navržen tak, aby byl co nejpřehlednější a srozumitelný. V budoucnu tak nebude problém jej rozšířit o další funkcionalitu. Prostor pro zlepšení se nachází například v možnosti jednodušeji zadávat velikost tohoto výpočetního okna bez nutnosti kompilace a možnosti dynamické volby tvaru kernelu v jednotlivých fázích výpočtu.

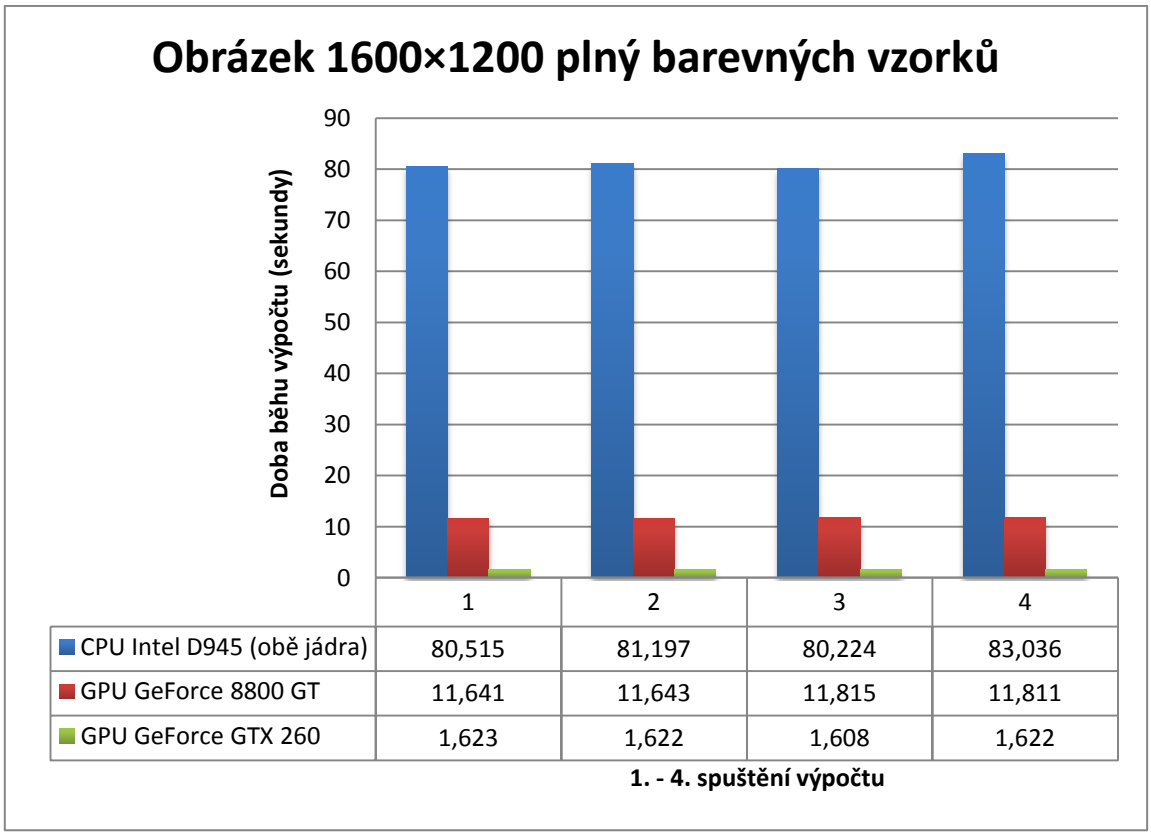
Algoritmus má hranici pro velikost výpočetního okna, která je dána maximálním počtem 512 vláken pro jeden blok. Pokud bychom chtěli zpracovávat výpočetní okna obsahující větší počet pixelů než tato hodnota, bylo by nutné přepracovat návrh algoritmu.

6.3 Grafické vyjádření konkrétních výsledků

Následující graf 1 ukazuje přímé porovnání výsledků pro obrázek 35 velikosti 1600×1200 pixelů, který byl téměř plný barevných vzorků, které musel algoritmus do výpočtu zahrnout. Měření doby běhu bylo provedeno čtyřikrát pro každý způsob zpracování, aby byly vidět odchylky v jednotlivých dobách běhu na stejné výpočetní jednotce.

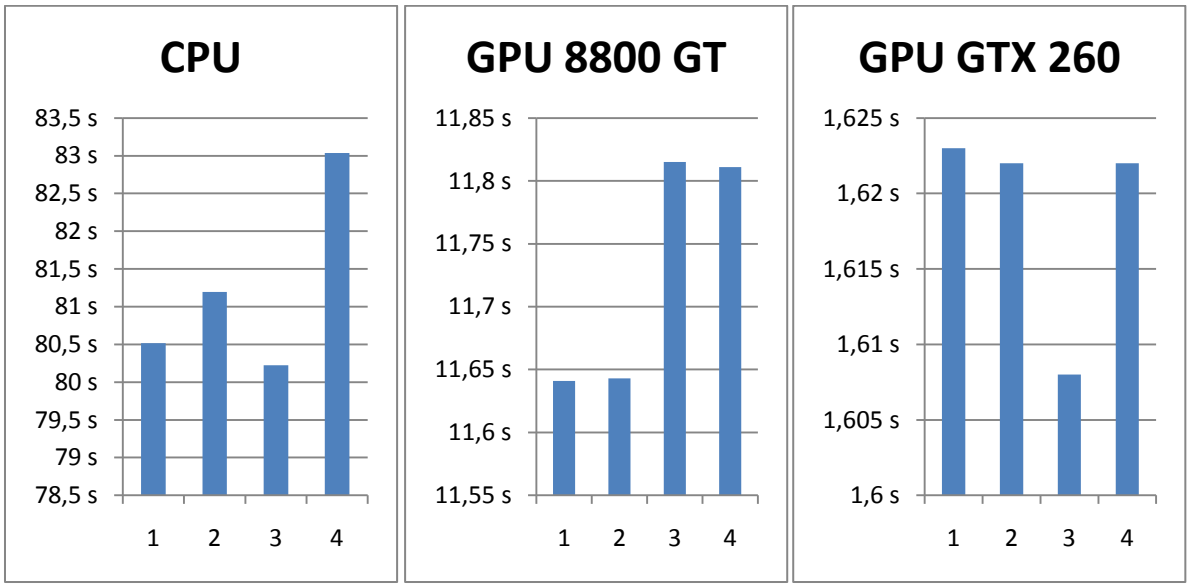


35 - Zpracovávaný obrázek 1



1 - Hustě obsazený velký obrázek

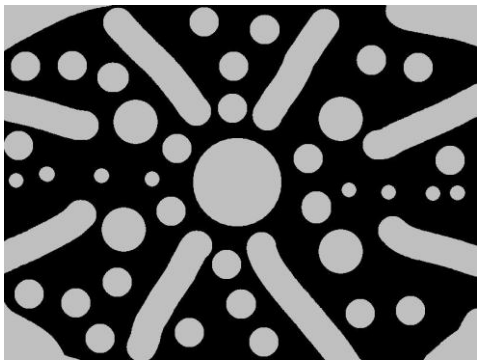
Na takto velkém obrázku se naplno projevila převaha paralelního zpracování. Ve čtvrtém spuštění byla moderní grafická karta GeForce GTX 260 schopna čas doby běhu výpočtu zkrátit na více než jednu padesátinu doby, kterou potřeboval pro zpracování naplno vytížený dvoujádrový procesor. U CPU jsou také více viditelné rozdíly doby běhu výpočtu, na rozdíl od zpracování na GPU, kde byly změny znatelně menší, zvláště pak u karty GTX 260. Průměr naměřených časů je pro CPU 81,243 sekund, pro kartu 8800 GT 11,7275 sekund a pro kartu GTX 260 1,61875 sekund. Směrodatná odchylka pro doby běhu na CPU je 1,093768 sekund, pro kartu 8800 GT 85,515 milisekund a pro kartu GTX 260 6,22 milisekund.



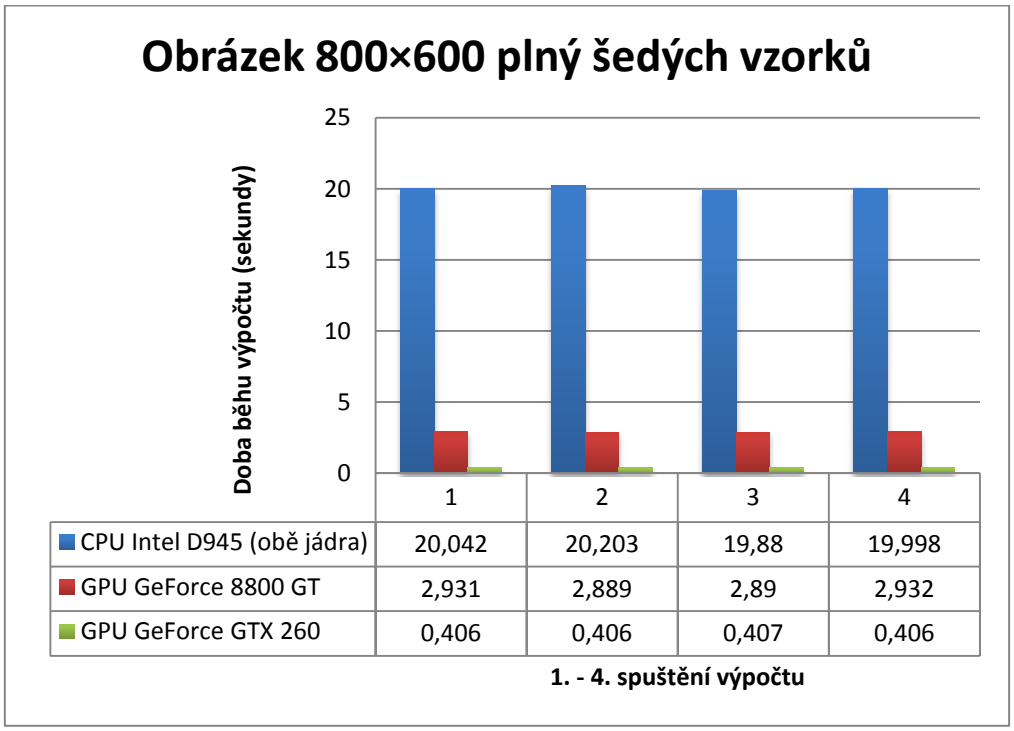
2 - Trojice grafů pro porovnání odchylek dob běhu

Uvedená trojice grafů přehledně znázorňuje odchylky v dobách běhu pro jednotlivé výpočetní jednotky při zpracování předchozího obrázku 35. U CPU odchylky dosahují řádově až několika sekund, což může v aplikacích závislých na konstantním výkonu znamenat problém. U současné grafické karty GeForce GTX 260 se odchylky pohybují v desítkách milisekund, což považuji u takto velkého obrázku za akceptovatelné.

Na dalším grafu 3 jsou znázorněny výsledné doby běhu výpočtu pro obrázek 36 velikosti 800×600 pixelů. Obrázek 36 je téměř plný šedých vzorků stejné barvy. Algoritmus opět do výpočtu zahrnul všechny tyto vzorky.

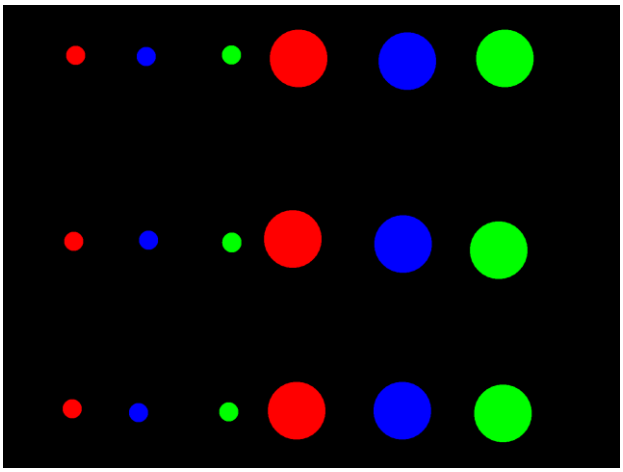


36 - Zpracovávaný obrázek 2

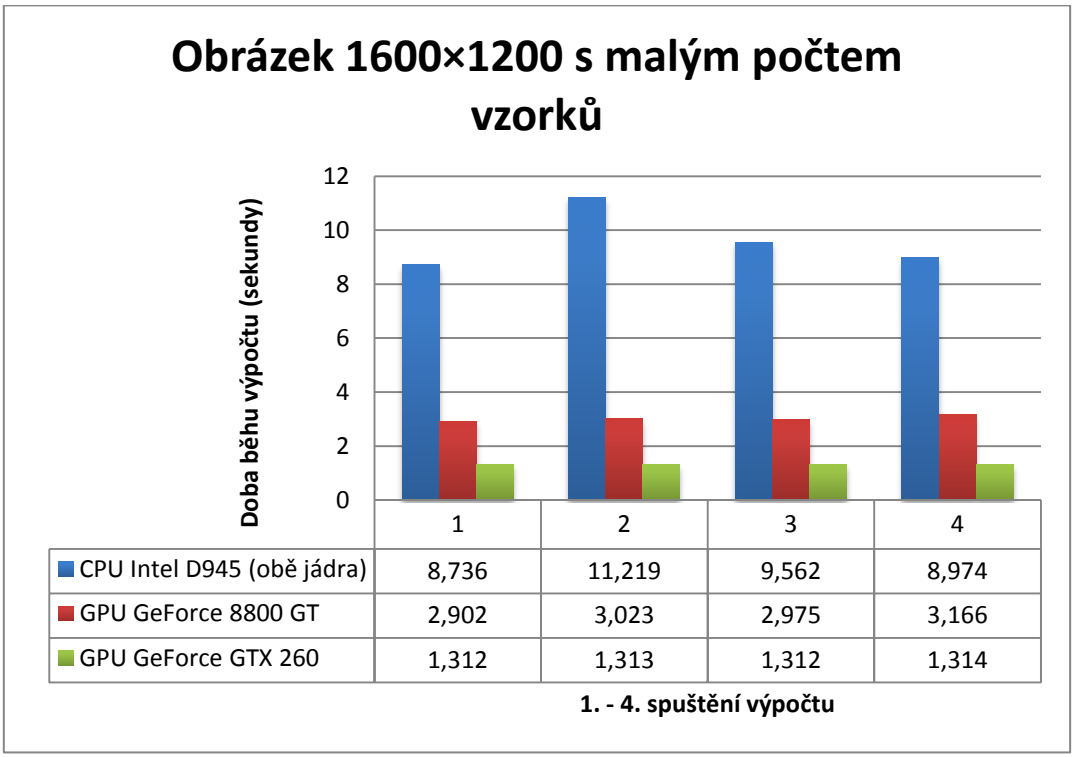


3 - Hustě obsazený střední obrázek

Průměr doby pro CPU je zde 20,03075 sekund, pro GPU 8800 GT 2,9105 sekund a pro GPU GTX 260 406,25 milisekund. Směrodatná odchylka pro doby běhu na CPU je 0,115753 sekund, pro kartu 8800 GT 21,006 milisekund a pro kartu GTX 260 0,433 milisekund. Již při prvním pohledu na tento graf je patrné, že dosažené výsledky jsou poměrně stejné, jako v minulém případě, z čehož jasně vyplývá fakt, že doba běhu výpočtu algoritmů není nijak závislá na barvě zpracovávaných vzorků, nýbrž na jejich počtu. Tuto situaci dokládá následující graf 4, kde je patrné, že při malém procentuálním zastoupení vzorků v obrázku 37 se rozdíly ve výkonu zmenší. I přesto je však paralelní zpracování na CUDA výrazně rychlejší.



37 - Zpracovávaný obrázek 3

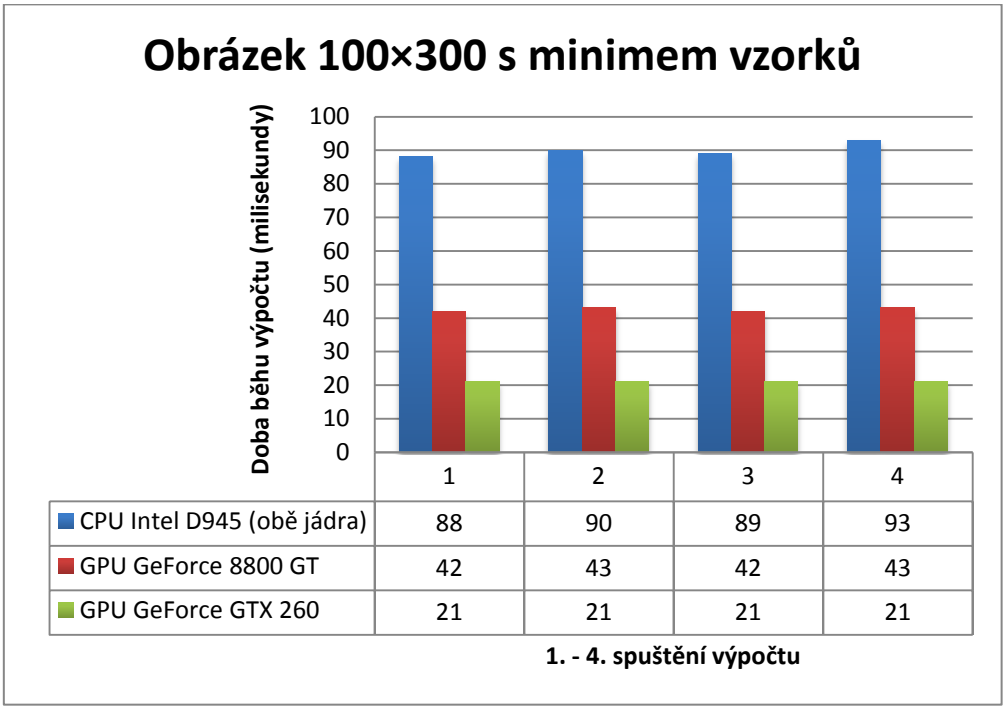


4 - Velký obrázek s málo vzorky

Průměr doby pro CPU je pro graf 4 9,62275 sekund, pro GPU 8800 GT 3,0165 sekund a pro GPU GTX 260 1,31275 sekund. Směrodatná odchylka pro doby běhu na CPU je 0,969395 sekund, pro kartu 8800 GT 96,469 milisekund a pro kartu GTX 260 0,829 milisekund.

Graf 5 v této kapitole ukazuje situaci při zpracovávání malého obrázku 38 velikosti 100×300 pixelů s minimálním počtem vzorků. Průměr naměřených časů je pro CPU 90 milisekund, pro kartu 8800 GT 42,5 milisekund a pro kartu GTX 260 21 milisekund. Směrodatná odchylka pro doby běhu na CPU je 1,870829 milisekund, pro kartu 8800 GT 0,5 milisekund. Karta GTX 260 vždy výpočet dokončila za konstantní dobu, tudíž neměla žádnou odchylku. Paralelní zpracování na moderní grafické kartě zde dosáhlo nejnižšího násobku výkonu oproti CPU. I pro takto malé zatížení však stále zůstává CUDA více než čtyřikrát rychlejší.

38 - Zpracovávaný obrázek 4



	1	2	3	4
CPU Intel D945 (obě jádra)	88	90	89	93
GPU GeForce 8800 GT	42	43	42	43
GPU GeForce GTX 260	21	21	21	21

5 - Malý obrázek s minimem vzorků

Poslední graf 5 této kapitoly zachycuje doby běhu potřebné k dokončení výpočtu pro různá nastavení velikosti výpočetního okna. Z obrázku lze poznat prudký nárůst doby zpracování na CPU, který je při zpracování na GPU řádově menší, což je jeho nespornou výhodou. Naměřené výsledky pro GPU se týkají modernější testované grafické karty GeForce GTX 260.

6 - Graf závislosti doby běhu na velikosti výpočetního okna

6.4 Shrnutí

U programu využívajícím pro svou práci CPU je vidět, že několik úrovní vnořených cyklů je na zpracování objemných dat nevhodné. Kvůli tomuto návrhu programu do hloubky také doba běhu procesorového programu stoupá výrazně rychleji, než u programu pro technologii CUDA, který se snaží zátěž přenést co nejvíce do šířky. Škálovatelnost je u programu na CUDA mnohem lepší, což dokládají rozdílné výsledky na dvou testovaných grafických kartách. Při použití více grafických karet s technologií CUDA na jednom počítači by bylo možné výkon dále násobit.

Mnou navržený algoritmus dosahuje oproti procesorovému řešení mnohonásobně kratší doby běhu výpočtu pro jednu iteraci se stejnými parametry. Největší podíl na zkrácení doby běhu programu nese paralelizace součtů polí s váženými parametry, která ušetří v celém programu největší část času. Zpracování programu dále značně zrychlují veškeré úkony, které jsou namísto původních cyklů realizovány paralelně. Konkrétně se jedná o paralelní vynulovávání polí a jejich paralelní plnění vypočtenými hodnotami. Na konci iterace jsou také paralelně překopírovány staré pozice, vypočteny pozice nové a následně je z těchto dvou pozic paralelně spočítán jejich rozdíl.

Z těchto důvodů vidím v úpravách a dalším budoucím vývoji tohoto CUDA algoritmu velkou perspektivu. V době, kdy jsem měl program již dlouho hotov a pracoval na této práci, byla vydána nová verze celého CUDA Toolkitu, která jistě odstraňuje spoustu chyb a nabízí vývojáři nové možnosti. Rovněž celý Nsight urazil za dobu mé práce velký kus cesty a je nyní k dostání v kompletní nové verzi. Dokážu si tedy představit, že by s využitím aktuálních nástrojů pro práci s architekturou CUDA bylo možné lépe zjišťovat příčiny chyb v programu, nebo že by se mnou popsané chyby nemusely dokonce vůbec objevit. Pro vývoj svého programu jsem používal beta verzi balíku Nsight z ledna 2010.

7 Závěr

Cílem této práce bylo navrhnout a vytvořit program pro výpočet realizující segmentaci pomocí metody Mean Shift, který by využíval nové technologie paralelního zpracování na architektuře CUDA. Program, který jsem pro tuto architekturu navrhl a implementoval, využívá mnoho výhod paralelního zpracování, což se v programu kladně projevilo. Především se jedná o výrazně kratší dobu běhu programu, kterou se mi podařilo zkrátit na zlomek doby, kterou pro stejný úkol vyžaduje existující řešení napsané pro práci s procesorem.

Můj program poskytuje efektivní a funkční základ, který již nyní dosahuje dobrých výsledků, pro další vývoj. V budoucnu tak nebude problém odstranit chyby, které jsem v rámci této práce popsal, a rozšířit funkčnost programu o výpočet dalších iterací. Vzhledem k tomu, že za dobu mé práce vyšly nové nástroje pro práci s architekturou CUDA, jmenovitě CUDA Toolkit 3 a nová verze balíku NVIDIA Parallel Nsight, bude další vývoj programu snazší a efektivnější.

Tato bakalářská práce je mým prvním projektem, který ve velké míře využívá paralelní zpracování jako takové, což mi dalo hodně zkušeností, které v budoucnu určitě využiji. Díky této práci jsem se do hloubky seznámil s nejnovější technologií CUDA, která má do budoucna zajisté velkou perspektivu, a naučil se používat nejnovější nástroje s touto technologií související. Hlavním přínosem je pro mě to, že jsem získal povědomí o paralelizaci programů. Jsem tak schopen si abstrakci paralelismu lépe představit, což mi dává možnost navrhnout algoritmus tak, aby možností paralelního zpracování efektivně využíval.

8 Literatura

[1] NVIDIA. *NVIDIA CUDA : Programming Guide*. Version 2.3, 1.7.2009.

[2] COMANICIU, Dorin; MEER, Peter. *IEEE Transactions on pattern analysis and machine inteligence* [online], 2002 [cit. 2010-05-03]. Mean Shift: A Robust Approach Toward Feature Space Analysis, Dostupné z WWW: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.4140&rep=rep1&type=pdf>.

[3] Kernel (statistics) In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 5.1.2007, 10.3.2010 [cit. 2010-05-03]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Kernel_(statistics)>.

[4] DOUBEK, Petr. Mean-Shift segmentace. 29.10.2007 [cit. 2010-05-03]. Dostupné z WWW: <http://cmp.felk.cvut.cz/cmp/courses/ZS1/Cviceni/cv4/meanshift.pdf>.

Přílohy

I. Výkonné jádro CUDA programu – kernel.cu

```
__global__ void kernel(
  int imageW,
  int imageH,
  const floatminmax *roi,
  const int kernelType,
  const float hxy,
  const float hcl,
  const float terminationPositionChange,
  const int maxNumIters,
  int *numAttrPixs,
  int *firstAttrPixs,
  int *attrPixs,
  float *posuny
){
__shared__ float vahy[441];
__shared__ float vazene_x[441];
__shared__ float vazene_y[441];
__shared__ float vazene_r[441];
__shared__ float vazene_g[441];
__shared__ float vazene_b[441];
__shared__ float sumy[6];
__shared__ float posOld[5];
__shared__ float pos[5];
__shared__ int maxi;
__shared__ bool b;

float inv_hxy2 = 1.0f/sqr(hxy);
float inv_hcl2 = 1.0f/sqr(hcl);
float4 buff;
int IDvlakna = threadIdx.y*blockDim.x+threadIdx.x;
float buffer[3];

int imageIndex = blockIdx.y*imageW+blockIdx.x;
if (IDvlakna==0){
    pos[0]=blockIdx.x;
    pos[1]=blockIdx.y;
    buff=tex2D(texImage, blockIdx.x, blockIdx.y);
    pos[2]=buff.x
    pos[3]=buff.y;
    pos[4]=buff.z;
    }
__syncthreads();

float index[2];
float g;

float buff_vazene_x;
float buff_vazene_y;
float buff_vazene_r;
float buff_vazene_g;
float buff_vazene_b;
float buff_vahy;

for (int itera=0; itera<1; itera++){

    vahy[IDvlakna] = 0.0f;
    vazene_x[IDvlakna] = 0.0f;
    vazene_y[IDvlakna] = 0.0f;
    vazene_r[IDvlakna] = 0.0f;
    vazene_g[IDvlakna] = 0.0f;
    vazene_b[IDvlakna] = 0.0f;
    if (IDvlakna<6) sumy[IDvlakna] = 0.0f;
    __syncthreads();

    index[0]=(threadIdx.x+pos[0]-(blockDim.x / 2));
    index[1]=(threadIdx.y+pos[1]-(blockDim.y / 2));

    if ( (index[0]>=0)&&(index[0]<imageW)&&(index[1]>=0)&&(index[1]<imageH) ){
```

```
        buff=tex2D(texImage, index[0], index[1]);
         buffer[0]=buff.x;
         buffer[1]=buff.y;
         buffer[2]=buff.z;

         if ( (roi[2][0]<=buffer[0])&&(buffer[0]<=roi[2][1])
                &&(roi[3][0]<=buffer[1])&&(buffer[1]<=roi[3][1])
                &&(roi[4][0]<=buffer[2])&&(buffer[2]<=roi[4][1]) ){

             g=inv_hxy2*(sqr(pos[0]-index[0]) +sqr(pos[1]-index[1]))
              +inv_hcl2*(sqr(pos[2]- buffer[0]) +sqr(pos[3]-buffer[1]) +sqr(pos[4]-buffer[2]));
             if (g <= 1.0f){
                 switch(kernelType){
                      case 0:
                        g = 1-sqrt(g);
                        break;
                      case 1:
                        g = 1-g;
                        break;
                      case 2:
                        g = 1-sqrt((sqrt(g)));
                        break;
                       }
              }
              else{
                  g=0.0f;
              }
              vahy[IDvlakna]=g;
          }
          else{
              vahy[IDvlakna]=0.0f;
          }
    else{
        vahy[IDvlakna]=0.0f;
    }
__syncthreads();

vazene_x[IDvlakna] = index[0]*vahy[IDvlakna];
vazene_y[IDvlakna] = index[1]*vahy[IDvlakna];
vazene_r[IDvlakna] = buffer[0]*vahy[IDvlakna];
vazene_g[IDvlakna] = buffer[1]*vahy[IDvlakna];
vazene_b[IDvlakna] = buffer[2]*vahy[IDvlakna];

if(threadIdx.x==0){
    maxi=220;
    b=true;
}
__syncthreads();

while(true){
    if (IDvlakna<maxi){
        if((b)&&(IDvlakna+1==maxi)){
        buff_vazene_x=vazene_x[(IDvlakna*2)]+vazene_x[(IDvlakna*2)+1]+vazene_x[(IDvlakna*2)+2];
        buff_vazene_y=vazene_y[(IDvlakna*2)]+vazene_y[(IDvlakna*2)+1]+vazene_y[(IDvlakna*2)+2];
        buff_vazene_r=vazene_r[(IDvlakna*2)]+vazene_r[(IDvlakna*2)+1]+vazene_r[(IDvlakna*2)+2];
        buff_vazene_g=vazene_g[(IDvlakna*2)]+vazene_g[(IDvlakna*2)+1]+vazene_g[(IDvlakna*2)+2];
        buff_vazene_b=vazene_b[(IDvlakna*2)]+vazene_b[(IDvlakna*2)+1]+vazene_b[(IDvlakna*2)+2];
        buff_vahy=vahy[(IDvlakna*2)]+vahy[(IDvlakna*2)+1]+vahy[(IDvlakna*2)+2];
        b=false;
        }else{
            buff_vazene_x = vazene_x[(IDvlakna*2)]+vazene_x[(IDvlakna*2)+1] ;
            buff_vazene_y = vazene_y[(IDvlakna*2)]+vazene_y[(IDvlakna*2)+1] ;
            buff_vazene_r = vazene_r[(IDvlakna*2)]+vazene_r[(IDvlakna*2)+1] ;
            buff_vazene_g = vazene_g[(IDvlakna*2)]+vazene_g[(IDvlakna*2)+1] ;
            buff_vazene_b = vazene_b[(IDvlakna*2)]+vazene_b[(IDvlakna*2)+1] ;
            buff_vahy = vahy[(IDvlakna*2)]+vahy[(IDvlakna*2)+1] ;
        }
    }

    if(IDvlakna==0){
        if (maxi%2!=0) b=true;
        maxi=maxi/2;
    }
    __syncthreads();

    vazene_x[IDvlakna]=buff_vazene_x;
    vazene_y[IDvlakna]=buff_vazene_y;
```

```
    vazene_r[IDvlakna]=buff_vazene_r;
    vazene_g[IDvlakna]=buff_vazene_g;
    vazene_b[IDvlakna]=buff_vazene_b;
    vahy[IDvlakna]=buff_vahy;

    if (maxi==0) break;
    }
__syncthreads();

if (IDvlakna<6){
    switch (IDvlakna){
               case 0: sumy[0]=vazene_x[0];
                       break;
               case 1: sumy[1]=vazene_y[0];
                       break;
               case 2: sumy[2]=vazene_r[0];
                       break;
               case 3: sumy[3]=vazene_g[0];
                       break;
               case 4: sumy[4]=vazene_b[0];
                       break;
               case 5: sumy[5]=vahy[0];
                       break;
    }
 }
__syncthreads();


if ( (sumy[5]>0.0f) && (IDvlakna<5) ){

    posOld[IDvlakna]=pos[IDvlakna];
    __syncthreads();

    pos[IDvlakna]=sumy[IDvlakna]/sumy[5];
    __syncthreads();

    vazene_x[IDvlakna] = sqr(pos[IDvlakna]-posOld[IDvlakna]);
    __syncthreads();

    if (IDvlakna==0){
        float ps = 0.0f;
        for (int h=0; h<5; h++) ps += vazene_x[h];
        posuny[imageIndex] =sqrt(ps);
    }
 }
__syncthreads();

}

}
```